Mizar System

December 10, 2023

Contents

Part I

Kernel

Chapter 1

Analyzer

Modules (called "`unit`"s in Pascal) have two parts: the public-facing "`interface`" declarations, and the private "`implementation`" code.

3a ⟨*kernel/analyzer.pas* 3a⟩≡

```
// The ANALYZER_REPORT ifdefs are left here, even though there is
// no other output. This is because the ifdefs now mark the output
// code precisely, which will be useful when rewriting it for XML.

{$DEFINE ANALYZER_REPORT}

unit analyzer;
interface
⟨Analyzer interface 3b⟩

implementation
⟨Analyzer implementation 3c⟩
end;
```

Root chunk (not used in this document).

The analyzer module has two procedures, `Analyze` and `DisposeAnalyze`.

3b ⟨*Analyzer interface* 3b⟩≡

```
procedure Analyze;
procedure DisposeAnalyze;

var  Verifying: boolean = true;

{$IFDEF FRM2THESIS}
var inConclusion  : boolean = false;
var inSchemeInfer : boolean = false;
{$ENDIF}
```

This code is used in chunk 3a.
Uses `Analyze` 138 and `DisposeAnalyze` 137b.

The implementation imports some common libraries with the `uses` keyword, declares some types, defines some constants, then defines a large number of functions.

3c ⟨*Analyzer implementation* 3c⟩≡

```
⟨Import common libraries for analyzer 4a⟩

⟨Declare types for analyzer 4b⟩

{+------------------------------------------------------------------------+}
⟨Define constants for analyzer 4c⟩

⟨Declare state variables for analyzer 4d⟩

⟨Analyzer methods 5a⟩
```

This code is used in chunk 3a.

4a ⟨*Import common libraries for analyzer* 4a⟩≡

```
uses lexicon,mconsole,limits,iocorrel,correl,mobjects,generato,identify,
     errhan,inout,mizenv,librenv,builtin,justhan,express,numbers,
     enums,formats,xml_parser,xmldict,xmlpars,mscanner
{$IFDEF ANALYZER_REPORT},inlibr,outlibr,inoutmml{$ENDIF}
{$IFDEF SKLTTEST},comact,edt_han{$ENDIF}
{$IFDEF MDEBUG},info,outinfo,absinfo{$ENDIF};
```

This code is used in chunk 3c.

4b ⟨*Declare types for analyzer* 4b⟩≡

```
type
   DefNodePtr = ^DefNode;
   DefNode =
      object(MObject)
         nMeansOccurs: char;
         nConstructor: Lexem; {'R','M','K','V',':'}
         SkIt,SkId,SkLabId,SkVarNbr: integer;
         DDef: DefPtr;
         nPrefix: RSNENTRY;
         nEssentials: IntSequence;
         nPrimaryList: MCollection;
         constructor
            Init(fMeansOccurs,fKind: char;
                 fLab,fLabId: integer;
                 fDef: DefPtr;
                 fEntry: RSNENTRY);
      end;
```

This code is used in chunk 3c.
Defines:
`DefNode`, used in chunk 29a.
`DefNodePtr`, used in chunks 29b, 117c, 119, and 121.

The constants defined are mostly error conditions.

4c ⟨*Define constants for analyzer* 4c⟩≡

```
const
   errFieldHomonimy = 91;
   errFieldTypeInconsistent = 92;
   errIncompletePrefix = 93;
   errNonStructPrefix = 94;
   CorrCondNbr=6;  // ##TODO: it seems that only 1..5 are used, not 6
```

This code is used in chunk 3c.

4d ⟨*Declare state variables for analyzer* 4d⟩≡

```
var
   RedefAntonym,gRedef,gSpecified,gPropertiesOcc: boolean;
   ResNbr: integer;
   AnyTyp: TypPtr;
   gProperties:PropertiesRec;
   gStatusOfProperties:integer;
   gDefiniens: DefPtr;
   gDefPos: Position;
   gSuperfluous,dPrimLength,gWhichOne: integer;
   ConstNr: array[1..2*MaxArgNbr] of integer;
   LocusAsConst: array[1..2*MaxArgNbr] of integer;
   gPrimaries: array[1..2*MaxArgNbr] of TypPtr;
   LociOcc: array[1..2*MaxArgNbr] of boolean;
   gNonPermissive: boolean;
   gPrimNbr,
   gBoundInc, { o ile inkrementowac zwiazane }
   gBoundForFirst, gBoundForSecond, gBoundForIt:integer;
```

```
{ przez jakie zmienne kwantyfikowane, argumenty maja byc zastapione }
gCorrCond: array[0..CorrCondNbr] of FrmPtr;
{$IFDEF ANALYZER_REPORT}
AReport:OutVRFFileObj;
{$ENDIF}
```

This code is used in chunk 3c.

1.1 Methods

5a ⟨*Analyzer methods* 5a⟩≡
⟨*Renew primaries* 5b⟩
⟨*Set Loci occurrences* 5c⟩
⟨*Change to constructor* 6a⟩
⟨*Renew Const* 6b⟩
⟨*Formal arguments* 6c⟩
⟨*Constructor formal arguments* 6d⟩
⟨*Read a sentence* 7a⟩
⟨*Read type* 7b⟩
⟨*Analyze term* 7c⟩
⟨*Analyze argument type list* 8b⟩
⟨*Read propositions* 9a⟩
⟨*Conjugate a list of propositions* 9b⟩
⟨*Scheme body* 9c⟩
⟨*Open a definition* 12a⟩
⟨*Close a definition* 12b⟩
⟨*Change fixed variables to bound variables* 12c⟩
⟨*xFormula(?)* 13a⟩
⟨*Get qualified list* 13b⟩
⟨*Get constant qualified list* 14a⟩
⟨*Write qualified* 14b⟩
⟨*Append locus* 15a⟩
⟨*Parameter declaration* 15b⟩
⟨*Change bound variable and iterate* 15c⟩
⟨*Make list of loci variables* 16a⟩
⟨*Check for compatible arguments* 16b⟩
⟨*Correctness conditions* 17b⟩

This definition is continued in chunks 25, 103a, and 114a.
This code is used in chunk 3c.

This is part of cleanup when restoring the state after declaring a structure type.

5b ⟨*Renew primaries* 5b⟩≡

```
procedure RenewPrimaries(fPrevLength:integer);
var
   k : integer;
begin
   for k := fPrevLength + 1 to gPrimNbr do dispose(gPrimaries[k],Done);
   gPrimNbr := fPrevLength;
   dPrimLength := fPrevLength;
end;
```

This code is used in chunk 5a.
Defines:
RenewPrimaries, used in chunk 64.

Within a definition, we can set the loci occurrences.

5c ⟨*Set Loci occurrences* 5c⟩≡

```
procedure SetLociOcc(var fTrm:TrmPtr);
begin
 with VarTrmPtr(fTrm)^ do if TrmSort=ikTrmLocus then LociOcc[VarNr] := true;
end;
```

This code is used in chunk 5a.
Defines:
SetLociOcc, used in chunks 31–33 and 59.

6a ⟨*Change to constructor* 6a⟩≡
```
procedure ChangeToConst(var fTrm : TrmPtr);
begin
   with VarTrmPtr(fTrm)^ do if TrmSort = ikTrmLocus then
   begin
      TrmSort := ikTrmConstant;
      VarNr := gSuperfluous+ConstNr[VarNr]
   end;
end;
```
This code is used in chunk 5a.
Defines:
ChangeToConst, used in chunks 39a and 64.

6b ⟨*Renew Const* 6b⟩≡
```
procedure RenewConst(var fTrm : TrmPtr);
begin
   with VarTrmPtr(fTrm)^ do
      if (TrmSort=ikTrmConstant) and (VarNr > g.VarNbr) then
         VarNr:=g.VarNbr;
end;
```
This code is used in chunk 5a.
Defines:
RenewConst, used in chunk 64.

This appears to produce a list of formal arguments (parameters?) to a term or type.

6c ⟨*Formal arguments* 6c⟩≡
```
function FormalArgs(fNbr:integer):TrmList;
var
   i: integer;
   lTL: TrmList;
begin
   lTL:=nil;
   for i:=fNbr downto 1 do lTL:=NewTrmList(NewVarTrm(ikTrmLocus,i),lTL);
   FormalArgs:=lTL;
end;
```
This code is used in chunk 5a.
Defines:
FormalArgs, used in chunks 63a and 64.

6d ⟨*Constructor formal arguments* 6d⟩≡
```
function C_FormalArgs(fNbr : integer) : TrmList;
var
   i: integer;
   lTL: TrmList;
begin
   lTL := nil;
   for i := fNbr downto 1 do lTL := NewTrmList(NewVarTrm(ikTrmConstant,i),lTL);
   C_FormalArgs := lTL;
end;
```
This code is used in chunk 5a.
Defines:
C_FormalArgs, used in chunk 39b.

Read in a sentence, produce a pointer to the formula's abstract syntax tree.

7a ⟨*Read a sentence* 7a⟩≡
```
function ReadSentence(Negate: boolean) : FrmPtr;
var
   lSnt : ExpPtr;
   lFrm : FrmPtr;
begin
   BoundVarNbr := 0;
   lSnt := LoadFormula;
   lFrm := lSnt^.Analyze;
   dispose(lSnt,Done);
   if Negate then lFrm := NewNegDis(lFrm);
   ReadSentence := lFrm;
end;
```
This code is used in chunk 5a.
Defines:
ReadSentence, used in chunks 9, 27a, 97, 100, and 135b.
Uses Analyze 138.

Read in a type, and return a pointer to the abstract syntax tree.

7b ⟨*Read type* 7b⟩≡
```
function ReadType : TypPtr;
var
   lExpPtr : ExpPtr;
   lTyp : TypPtr;
begin
   BoundVarNbr  :=  0;
   lExpPtr  :=  LoadType;
   lTyp  :=  lExpPtr^.Analyze;
   ReadType  :=  lTyp;
   dispose(lExpPtr,Done);
end;
```
This code is used in chunk 5a.
Defines:
ReadType, used in chunks 10, 13b, 14a, 38a, 39a, 47b, 52–54, 62a, 64, 100, and 114b.
Uses Analyze 138.

Analyze an expression pointer as a term pointer.

7c ⟨*Analyze term* 7c⟩≡
```
function AnalyzeTerm(aExpr : ExpPtr): TrmPtr;
var
   lTrm,lTrm1: TrmPtr;
begin
   BoundVarNbr := 0;
   lTrm := aExpr^.Analyze;
   if lTrm^.TrmSort=ikTrmQua then
   begin
      lTrm1 := CopyTerm(QuaTrmPtr(lTrm)^.TrmProper);
      DisposeTrm(lTrm); lTrm := lTrm1;
   end;
   AnalyzeTerm := lTrm;
end;
```
This code is used in chunk 5a.
Defines:
AnalyzeTerm, used in chunk 27c.
Uses Analyze 138.

8a ⟨*Read a term* 8a⟩≡

```
function ReadTerm : TrmPtr;
var
   lTrm,lTrm1: TrmPtr;
   lExpPtr: ExpPtr;
begin
   BoundVarNbr := 0;
   lExpPtr := LoadTerm;
   lTrm := lExpPtr^.Analyze;
   dispose(lExpPtr,Done);
   if lTrm^.TrmSort = ikTrmQua then
   begin
      lTrm1 := CopyTerm(QuaTrmPtr(lTrm)^.TrmProper);
      DisposeTrm(lTrm);
      lTrm := lTrm1;
   end;
   ReadTerm := lTrm;
end;
```

Root chunk (not used in this document).
Defines:
`ReadTerm`, used in chunks 78c, 88b, 97, and 100.
Uses `Analyze` 138.

8b ⟨*Analyze argument type list* 8b⟩≡

```
var gMaxArgNbr: integer = MaxArgNbr;
// 'gMaxArgNbr' is changed in the 'REgistration' procedure to '2*MaxArgNbr'
// for an identify registration only.
// In an identify registration two pattern are occuring.
// Two pattern can have 'MaxArgNbr' arguments each one.
// Still the number of locus must be '2*MaxArgNbr' in parameters list as
// a maximal number of locus. The 'MaxArgNbr' locuses for one of the two patterns.
// The 'MaxArgNbr' is a limit for number of arguments in any pattern.

procedure AnalizeArgTypeList(var fTypList : MList);
var
   n,z: integer;
   lColl: MCollection;
   lExpPtr:ExpPtr;
begin
   n := 0;
   lColl.Init(2,2);
   inFile.InWord;
   while InFile.Current.Kind <> ';' do
   begin
      lExpPtr := LoadType;
      lColl.Insert(lExpPtr);
      InFile.InWord;
   end;
   fTypList.Init(lColl.Count);
   with lColl do
      for z := 0 to Count-1 do
      begin
         if n >= gMaxArgNbr then OverflowError(937);
         inc(n);
         BoundVarNbr := 0;
         LocArgTyp[n] := ExpPtr(Items^[z])^.Analyze;
         fTypList.Insert(LocArgTyp[n]);
      end;
   lColl.Done;
end;
```

This code is used in chunk 5a.
Defines:
AnalizeArgTypeList, used in chunks 10, 11, and 100.
gMaxArgNbr, used in chunks 15, 114b, and 129.
Uses Analyze 138.

9a ⟨*Read propositions* 9a⟩≡
```
procedure ReadPropositions(var fConditions : MCollection);
var
   lFrm: FrmPtr;
   lLabNr,lLabId: integer;
   lPos: Position;
begin
   fConditions.Init(2,4);
   while InFile.Current.Kind <> ';' do
   begin
      lLabNr := InFile.Current.Nr;
      InFile.InInt(lLabId);
      InFile.InPos(lPos);
      InFile.InWord;
      lFrm := ReadSentence(false);
      fConditions.Insert(new(PropositionPtr,Init(lLabNr,lLabId,lFrm,lPos)));
      InFile.InWord;
   end;
end;
```
This code is used in chunk 5a.
Defines:
ReadPropositions, used in chunks 9c, 78c, 82, 88b, 92, 100, and 121.
Uses ReadSentence 7a.

9b ⟨*Conjugate a list of propositions* 9b⟩≡
```
function ConjugatePropositions(const fConditions : MCollection) : FrmPtr;
var
   lFrm:FrmPtr;
   z: integer;
begin
   lFrm := NewVerum;
   with fConditions do
      for z := 0 to Count-1 do
         lFrm := NewConj(lFrm,PropositionPtr(Items^[z])^.nSentence^.CopyFormula);
   ConjugatePropositions := lFrm;
end;
```
This code is used in chunk 5a.
Defines:
ConjugatePropositions, used in chunks 78c, 82, 88b, 92, 100, and 121.

9c ⟨*Scheme body* 9c⟩≡
```
var gSchemeThesis: FrmPtr;

var gSchPredNbr,CurSchFuncNbr: integer;
procedure SchemeBody;
var
   lSchVarNbr,k,j,lSchId: integer;
   lTypList: MCollection;
   lTyp: TypPtr;
   lConditions: MCollection;
begin
   InFile.InWord;
   InFile.InInt(lSchId);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart(elSchemeBlock);
   AReport.Out_PosAsAttrs(CurPos);
```

```
      AReport.Out_XIntAttr(atSchemeNr, InFile.Current.Nr);
      AReport.Out_XIntAttr(atVid, lSchId);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      InFile.InPos(CurPos);
      InFile.InWord;
      gSchPredNbr := 0;
      CurSchFuncNbr := 0;
      CurSchFuncTyp.Init(MaxFuncVarNbr,0);
      while InFile.Current.Kind<>';' do
         case InFile.Current.Kind of
            ikTrmSchFunc:
               ⟨Handle term scheme functor 10⟩
            ikFrmSchPred:
               ⟨Handle formula scheme predicate 11⟩
         else
         begin
            {$IFDEF MDEBUG}
            writeln(infofile,'InFile.Current.Kind=',InFile.Current.Kind);
            {$ENDIF}
            RunTimeError(2064);
         end;
         end;
      CurSchFuncTyp.SetLimit(0);
      InFile.InWord;
      gSchemeThesis := ReadSentence(false);
      InFile.InWord;
      ReadPropositions(lConditions);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elSchemePremises);
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elSchemePremises);
      {$ENDIF}
      lConditions.Done;
   end;
```

This code is used in chunk 5a.
Defines:
CurSchFuncNbr, used in chunks 10 and 134.
gSchemeThesis, used in chunk 134.
gSchPredNbr, used in chunks 11 and 134.
SchemeBody, used in chunk 134.
Uses ReadPropositions 9a and ReadSentence 7a.

10 ⟨*Handle term scheme functor 10*⟩≡

```
   begin
      lSchVarNbr := 0;
      while InFile.Current.Kind<>';' do
      begin
         inc(lSchVarNbr);
         SchFuncArity[CurSchFuncNbr+lSchVarNbr].nId := InFile.Current.Nr;
         InFile.InWord;
      end;
      Mizassert(2616,CurSchFuncNbr+lSchVarNbr <= MaxFuncVarNbr);
      AnalizeArgTypeList(lTypList);
      InFile.InWord; lTyp := ReadType;
      for k := 1 to lSchVarNbr do
      begin
         { azeby umozliwic dysponowanie, trzeba cala kolekcje skopiowac }
         with SchFuncArity[CurSchFuncNbr+k] do
         begin
            SchFuncArity[CurSchFuncNbr+k].nArity.Init(lTypList.Count);
```

```
            for j := 0 to lTypList.Count-1 do
               SchFuncArity[CurSchFuncNbr+k].nArity.
                  Insert(TypPtr(lTypList.Items^[j])^.CopyType);
         end;
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elSchemeFuncDecl);
         AReport.Out_XIntAttr(atNr, CurSchFuncNbr+k);
         AReport.Out_XIntAttr(atVid, SchFuncArity[CurSchFuncNbr+k].nId);
         AReport.Out_XAttrEnd;
         AReport.Out_ArgTypes(lTypList);
         AReport.Out_Type(lTyp);
         AReport.Out_XElEnd(elSchemeFuncDecl);
         {$ENDIF}
         CurSchFuncTyp.Insert(lTyp^.CopyType);
      end;
      lTypList.Done;
      dispose(lTyp,Done);
      inc(CurSchFuncNbr,lSchVarNbr);
      Infile.InWord;
   end;
```

This code is used in chunk 9c.
Uses AnalizeArgTypeList 8b, CurSchFuncNbr 9c, and ReadType 7b.

11 ⟨*Handle formula scheme predicate* 11⟩≡

```
   begin
      lSchVarNbr := 0;
      while InFile.Current.Kind<>';' do
      begin
         inc(lSchVarNbr);
         SchPredArity[gSchPredNbr+lSchVarNbr].nId := InFile.Current.Nr;
         InFile.InWord;
      end;
      Mizassert(2517,gSchPredNbr+lSchVarNbr <= MaxPredVarNbr);
      AnalizeArgTypeList(lTypList);
      for k := 1 to lSchVarNbr do
         with SchPredArity[gSchPredNbr+k] do
      begin
                     {$IFDEF ANALYZER_REPORT}
                     AReport.Out_XElStart(elSchemePredDecl);
                     AReport.Out_XIntAttr(atNr, gSchPredNbr+k);
                     AReport.Out_XIntAttr(atVid, SchPredArity[gSchPredNbr+k].nId);
                     AReport.Out_XAttrEnd;
                     AReport.Out_ArgTypes(lTypList);
                     AReport.Out_XElEnd(elSchemePredDecl);
                     {$ENDIF}
                     nArity.Init(lTypList.Count);
                     for j := 0 to lTypList.Count-1 do
                        SchPredArity[gSchPredNbr+k].nArity.
                           Insert(TypPtr(lTypList.Items^[j])^.CopyType);
                  end;
                  lTypList.Done;
                  inc(gSchPredNbr,lSchVarNbr);
                  InFile.InWord;
               end;
```

This code is used in chunk 9c.
Uses AnalizeArgTypeList 8b and gSchPredNbr 9c.

12a ⟨*Open a definition* 12a⟩≡
```
var D: LevelRec;

procedure OpenDef;
begin
   InFile.InWord;
   gNonPermissive := true;
   gPrimNbr := 0;
   dPrimLength := 0;
   gDefBase := g.VarNbr;
end;
```
This code is used in chunk 5a.
Defines:
D, used in chunks 121, 129, and 132.
OpenDef, used in chunks 121, 129, and 132.

12b ⟨*Close a definition* 12b⟩≡
```
procedure CloseDef;
var
   k: integer;
   nk: NotationKind;
begin
   for nk := Low(NotationKind) to High(NotationKind) do
      with Notat[nk] do
   begin
      {$IFDEF ANALYZER_REPORT}
      for k :=  Count to Count + fExtCount - 1 do
      begin
         if (nk = noMode) and (PatternPtr(Items^[k])^.Expansion <> nil) then
         begin
            PatternPtr(Items^[k])^.Expansion^.LowerCluster^.ClearPids;
            PatternPtr(Items^[k])^.Expansion^.UpperCluster^.ClearPids;
         end;
      end;
      {$ENDIF}
      Notat[nk].AddExtItems;
   end;
   gPrimNbr := 0;
   dPrimLength := 0;
   for k := 1 to gPrimNbr do
      dispose(gPrimaries[k],Done);
   RegisteredCluster.AddExtItems;
   ConditionalCluster.AddExtItems;
   FunctorCluster.AddExtItems;
end;
```
This code is used in chunk 5a.
Defines:
CloseDef, used in chunks 121, 129, and 132.

12c ⟨*Change fixed variables to bound variables* 12c⟩≡
```
var gFixedBase: integer;

procedure ChangeFixedToBound(var fTrm: TrmPtr);
begin
   with VarTrmPtr(fTrm)^ do
      case TrmSort of
         ikTrmBound:
            inc(VarNr, g.VarNbr - gFixedBase);
         ikTrmConstant:
            if VarNr > gFixedBase then
            begin
```

```
                TrmSort := ikTrmBound;
                dec(VarNr, gFixedBase)
            end;
        end;
    end;
```

This code is used in chunk 5a.
Defines:
ChangeFixedToBound, used in chunk 13a.
gFixedBase, used in chunks 13, 14, 75, 78c, 88b, and 114b.

The xFormula function appears to transform a formula into "Mizar normal form" as a semantic correlate.

The NewNegDis is a method defined in the semantic correlate library, which negates constructs a new formula and dispose the argument passed to it.

13a ⟨*xFormula(?)* 13a⟩≡

```
function xFormula(fForm: FrmPtr): FrmPtr;
var
   lTyp: TypPtr;
   kk: integer;
begin
   fForm := NewNegDis(fForm);
   for kk := g.VarNbr downto gFixedBase + 1 do
   begin
      lTyp := FixedVar[kk].nTyp;
      if lTyp^.TypSort=ikError then
      begin
         xFormula := NewInCorFrm;
         exit;
      end;
      fForm := NewUnivI(FixedVar[kk].nIdent, lTyp^.CopyType, fForm);
   end;
   WithInFormula(fForm, ChangeFixedToBound);
   xFormula := NewNegDis(fForm);
end;
```

This code is used in chunk 5a.
Defines:
xFormula, used in chunks 78c, 88b, 100, and 121.
Uses ChangeFixedToBound 12c and gFixedBase 12c.

13b ⟨*Get qualified list* 13b⟩≡

```
procedure GetQualifiedList;
var
   lNbr,i: integer;
   lTyp: TypPtr;
begin
   gFixedBase := g.VarNbr;
   InFile.InWord;
   while InFile.Current.Kind = 'Q' do
   begin
      lNbr := g.VarNbr;
      inc(g.VarNbr,InFile.Current.Nr);
      for i := 1 to InFile.Current.Nr do
      begin
         InFile.InWord; // 'I'
         FixedVar[lNbr+i].nIdent := InFile.Current.Nr;
      end;
      lTyp := ReadType;
      for i := lNbr + 1 to g.VarNbr do
      begin
         FixedVar[i].nSkelConstNr := 0;
         if i = g.VarNbr then
            FixedVar[i].nTyp := lTyp
```

```
            else
               FixedVar[i].nTyp := lTyp^.CopyType;
            FixedVar[i].nExp := false;
         end;
         // dispose(lTyp,Done);
         InFile.InWord;
      end;
   end;
```

This code is used in chunk 5a.
Defines:
GetQualifiedList, used in chunks 78c, 88b, and 100.
Uses gFixedBase 12c and ReadType 7b.

14a ⟨*Get constant qualified list* 14a⟩≡

```
procedure GetConstQualifiedList;
var
   lNbr,i: integer;
   lTyp: TypPtr;
begin
   gFixedBase := g.VarNbr;
   InFile.InWord;
   while InFile.Current.Kind = 'Q' do
   begin
      lNbr := g.VarNbr;
      inc(g.VarNbr,InFile.Current.Nr);
      for i := 1 to InFile.Current.Nr do
      begin InFile.InWord; // 'I'
      FixedVar[lNbr + i].nIdent := InFile.Current.Nr;
      end;
      lTyp := ReadType;
      for i := lNbr + 1 to g.VarNbr do
      begin
         FixedVar[i].nSkelConstNr := -1;
         if i=g.VarNbr then
            FixedVar[i].nTyp := lTyp
         else FixedVar[i].nTyp := lTyp^.CopyType;
         FixedVar[i].nExp := false;
      end;
      //   dispose(lTyp,Done);
      InFile.InWord;
   end;
end;
```

This code is used in chunk 5a.
Defines:
GetConstQualifiedList, used in chunks 64 and 121.
Uses gFixedBase 12c and ReadType 7b.

14b ⟨*Write qualified* 14b⟩≡

```
procedure WriteQualified;
var
   i: integer;
begin
   {$IFDEF ANALYZER_REPORT}
   for i := gFixedBase+1 to g.VarNbr do
      AReport.Out_TypeWithId(FixedVar[i].nTyp,FixedVar[i].nIdent);
   {$ENDIF}
end;
```

This code is used in chunk 5a.
Defines:
WriteQualified, used in chunks 78c, 88b, 100, 114b, and 121.
Uses gFixedBase 12c.

15a ⟨*Append locus* 15a⟩≡

```
procedure AppendLocus(fTyp: TypPtr);
begin
   inc(g.VarNbr);
   { brak kotroli OverFlow !!!!!!!!!!!!!! }
   FixedVar[g.VarNbr].nIdent := 0;
   FixedVar[g.VarNbr].nTyp := fTyp;
   FixedVar[g.VarNbr].nExp := false;
   if dPrimLength >= gMaxArgNbr then OverflowError(937);
   inc(dPrimLength);
   if gPrimNbr >= gMaxArgNbr then OverflowError(937);
   inc(gPrimNbr);
   inc(g.GenCount);
   FixedVar[g.VarNbr].nSkelConstNr := g.GenCount;
   LocusAsConst[g.GenCount] := g.VarNbr;
   ConstNr[gPrimNbr] := g.VarNbr;
   gPrimaries[gPrimNbr] := AdjustedType(fTyp);
   gPrimaries[gPrimNbr]^.WithinType(ChangeToLoci)
end;
```

This code is used in chunk 5a.
Defines:
`AppendLocus`, used in chunk 64.
Uses `gMaxArgNbr` 8b.

15b ⟨*Parameter declaration* 15b⟩≡

```
procedure ParamDecl(fVarBase: integer);
var
   i: integer;
begin
   dPrimLength := g.GenCount + g.VarNbr - fVarBase;
   if dPrimLength>gMaxArgNbr then OverflowError(479);
   for i := fVarBase + 1 to g.VarNbr do
   begin
      if gPrimNbr >= gMaxArgNbr then OverflowError(937);
      inc(gPrimNbr);
      inc(g.GenCount);
      FixedVar[i].nSkelConstNr := g.GenCount;
      LocusAsConst[g.GenCount] := i;
      ConstNr[gPrimNbr] := i;
      gPrimaries[gPrimNbr] := AdjustedType(FixedVar[i].nTyp);
      gPrimaries[gPrimNbr]^.WithinType(ChangeToLoci);
   end;
end;
```

This code is used in chunk 5a.
Defines:
`ParamDecl`, used in chunks 64 and 114b.
Uses `gMaxArgNbr` 8b.

15c ⟨*Change bound variable and iterate* 15c⟩≡

```
procedure ChangeBoundAndIt(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
   with VarTrmPtr(fTrm)^ do
      case TrmSort of
         ikTrmIt:
            begin
               lTrm := fTrm;
               fTrm := NewVarTrm(ikTrmBound,1);
               dispose(lTrm,Done);
            end;
```

```
            ikTrmBound: inc(VarNr);
         end;
   end;
```

This code is used in chunk 5a.
Defines:
ChangeBoundAndIt, used in chunks 18–20.

16a ⟨*Make list of loci variables* 16a⟩≡

```
   function LociList(fLength: integer): TrmList;
   var
      lTrmList: TrmList;
      k: integer;
   begin
      lTrmList := nil;
      for k := fLength downto gSuperfluous+1 do
         lTrmList := NewTrmList(NewVarTrm(ikTrmConstant,LocusAsConst[k]),lTrmList);
      LociList := lTrmList;
   end;
```

This code is used in chunk 5a.
Defines:
LociList, used in chunks 18a and 106b.

16b ⟨*Check for compatible arguments* 16b⟩≡

```
   { ComaptibleArgs, ktore tak naprawde sa identyfikacja oryginalu, sa
     dobrym miejscem, zeby skonstruowac liste argumentow definiendum.
   }
   { Google translate:
     ComaptibleArgs, which are actually the identification of the original,
     are a good place to construct the definiendum argument list. }

   var gDefiniendumArgs: TrmList;
       gDefArgsError: boolean;

   function CompatibleArgs(L: integer): boolean;
   var
      i,S: integer;
      procedure CompError(fErrNr:integer);
      begin
         ErrImm(fErrNr);
         S := 0;
         CompatibleArgs := false;
         gDefArgsError := true;
      end;
   begin
      S := dPrimLength-L;
      gDefArgsError := false;
      if S < 0 then begin CompError(107); exit end;
      for i := 1 to L do
         with VarTrmPtr(gSubstTrm[i])^ do
            if (TrmSort=ikTrmConstant) and(FixedVar[VarNr].nSkelConstNr<>0) then
            begin
               if FixedVar[VarNr].nSkelConstNr<>S+i then begin CompError(109); exit end
            end
            else begin CompError(108); exit end;
      CompatibleArgs := true;
      {-----}
      {- gDefiniendumArgs := nil; -}
      for i := L downto 1 do
         gDefiniendumArgs := NewTrmList(CopyTerm(gSubstTrm[i]),gDefiniendumArgs);
   end;
```

This code is used in chunk 5a.

Defines:
CompatibleArgs, used in chunks 34c, 35, 39b, 41, 44, 45, 48, and 49.
gDefArgsError, used in chunk 18b.
gDefiniendumArgs, used in chunks 18b, 121, 129, and 132.

17a ⟨*Write definiens label* 17a⟩≡

```
var
   gDefNode:
      record
         MeansOccurs: char;
         Specified,Positive: boolean;
         Pos1,Pos2: Position;
         Kind: char;
         LabNr,LabId,Length: integer;
         fPrimaries: MCollection;
         { Poniewaz nie wiadomo, czy bedzie nowy konstruktor,
           trzeba je na razie tutaj uzbierac.
         }
         { Google translate:
           Since it is not known whether there will be a new constructor,
           they need to be collected here for now. }
      end;
   gConstErr: boolean;

{$IFDEF ANALYZER_REPORT}
procedure WriteDefiniensLabel;
begin
   if gDefNode.MeansOccurs<>' ' then
   begin
      AReport.Out_XIntAttr(atNr, gDefNode.LabNr);
      AReport.Out_XIntAttr(atVid, gDefNode.LabId);
      AReport.Out_PosAsAttrs(gDefNode.Pos2);
   end;
end;
{$ENDIF}
```

Root chunk (not used in this document).
Defines:
gConstErr, used in chunks 26a, 27c, 87b, and 88a.
gDefNode, used in chunks 27c, 29, 33–35, 38b, 39b, 41, 43–45, 47–49, 121, and 138.
WriteDefiniensLabel, used in chunk 121.

1.2 Correctness Conditions

Definitions have associated correctness conditions (except for attributes and predicates). These are analyzed in these functions.

17b ⟨*Correctness conditions* 17b⟩≡
⟨*Coherence* 18a⟩
⟨*Compatibility* 18b⟩
⟨*Consistency* 19⟩
⟨*Existence* 20⟩
⟨*Parse coherence in equals definition* 21⟩
⟨*Change bound variables and iterate (one)* 22a⟩
⟨*Copy formula with fresh bound variables* 22b⟩
⟨*Change bound variables and iterate (two)* 22c⟩
⟨*Partial uniqueness condition* 23a⟩
⟨*Parse uniqueness condition* 23b⟩

This code is used in chunk 5a.

18a ⟨*Coherence* 18a⟩≡

```
function Coherence(ff: char): FrmPtr;
var
   OldType: TypPtr;
   Sample: TrmPtr;
   cFrm: FrmPtr;
   lArgs: TrmList;
begin
   case ff of
      'M':
         with Notat[noMode] do
         begin
            lArgs := LociList(PatternPtr(Items^[Count+fExtCount-1])^.fPrimTypes.Count);
            with  ConstrTypPtr(Constr[coMode].Items^[gWhichOne])^ do
               OldType :=
               NewStandardTyp(ikTypMode,NewEmptyCluster,
                              InstCluster(fConstrTyp^.UpperCluster,lArgs),
                              gWhichOne,lArgs);
            Sample := NewVarTrm(ikTrmBound,1);
            cFrm := NewUniv(OldType,NewQualFrm(Sample,ItTyp^.CopyType));
         end;
      'K':
         with Notat[noFunctor] do
         begin
            lArgs := LociList(PatternPtr(Items^[Count+fExtCount-1])^.fPrimTypes.Count);
            Sample := NewFuncTrm(gWhichOne,lArgs);
            cFrm := NewQualFrm(Sample,ItTyp^.CopyType);
         end;
      else
         RunTimeError(2005);
   end;
   Coherence := cFrm;
end;
```

This code is used in chunk 17b.
Defines:
Coherence, used in chunks 39b and 44.
Uses LociList 16a.

18b ⟨*Compatibility* 18b⟩≡

```
function Compatibility(ff: char): FrmPtr;
var
   lDefiniendum: FrmPtr;
   function PartBiCond(fFrm: FrmPtr): FrmPtr;
   begin
      PartBiCond := NewBiCond(lDefiniendum^.CopyFormula,fFrm^.CopyFormula)
   end;
var
   z: integer;
   lOth,cFrm,lFrm: FrmPtr;
begin
   if (gDefiniens = nil) or gDefArgsError or (gWhichOne=0) then
   begin
      Compatibility := NewInCorFrm;
      exit;
   end;
   case ff of
      'M':
         lDefiniendum :=
         NewQualFrm(NewItTrm,
                    NewStandardTyp(ikTypMode,NewEmptyCluster,
                                   InstCluster(ItTyp^.UpperCluster,gDefiniendumArgs),
```

```
                                    gWhichOne,gDefiniendumArgs));
        'K':
           with ConstrTypPtr(Constr[coFunctor].Items^[gWhichOne])^.nPrimaries do
           lDefiniendum := NewEqFrm(NewItTrm,NewFuncTrm(gWhichOne,ReNewArgs(Count,gDefiniendumArgs)));
        'R': lDefiniendum := NewPredFrm(ikFrmPred,gWhichOne,gDefiniendumArgs,0);
        'V':
           with ConstrTypPtr(Constr[ coAttribute].Items^[gWhichOne])^.nPrimaries do
           lDefiniendum := NewPredFrm(ikFrmAttr,gWhichOne,ReNewArgs(Count,gDefiniendumArgs),0);
        else
        begin
           {$IFDEF MDEBUG}
           writeln(InfoFile,ff,'|');
           {$ENDIF}
           RunTimeError(2006);
        end;
     end;
     gDefiniendumArgs := nil;
     with gDefiniens^ do
     begin
        if nOtherwise <> nil then lOth := NewVerum;
        cFrm := NewVerum;
        with nPartialDefinientia do
           for z := 0 to Count-1 do
              with PartDefPtr(Items^[z])^ do
           begin
              if gDefiniens^.nOtherwise <> nil then
                 lOth := NewConj(lOth,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
              case DefSort of
                 'm': lFrm := FrmPtr(nPartDefiniens);
                 'e': lFrm := NewEqFrm(NewItTrm,TrmPtr(nPartDefiniens));
                 else RunTimeError(2503);
              end;
              cFrm := NewConj(cFrm,NewImpl(FrmPtr(nGuard)^.CopyFormula,PartBiCond(lFrm)));
           end;
        if nOtherwise <> nil then
        begin
           case DefSort of
              'm': lFrm := FrmPtr(nOtherwise);
              'e': lFrm := NewEqFrm(NewItTrm,TrmPtr(nOtherwise));
              else RunTimeError(2504);
           end;
           cFrm := NewConj(cFrm,NewImpl(lOth,PartBiCond(lFrm)));
        end
     end;
     if ff in ['K','M'] then
     begin
        WithInFormula(cFrm,ChangeBoundAndIt);
        cFrm := NewUniv(ItTyp^.CopyType,cFrm);
     end;
     dispose(lDefiniendum,Done);
     Compatibility := cFrm;
  end;
```

This code is used in chunk 17b.
Defines:
  Compatibility, used in chunk 27c.
Uses ChangeBoundAndIt 15c, gDefArgsError 16b, and gDefiniendumArgs 16b.

19 ⟨*Consistency* 19⟩≡

```
  function Consistency(ff: char): FrmPtr;
  var
     cFrm,cFrm2,EqFrm,lFrm1,lFrm2: FrmPtr;
```

```
    i,j: integer;
  begin
    {if gErrorInDefinition then begin Consistency := NewInCorFrm; exit end;
     watpliwe czy taki ogolny warunek ma sens, w koncu do sformulowania
     "consistency" potrzebujemy jedynie definiensu, a jezeli byly jakies
     niepoprawne zdania w definiensie, to chyba generowanie zdan powinno to
     zalatwic.
    }
    { Google translate:
     if gErrorInDefinition then begin Consistency := NewInCorFrm; exit end;
     it is doubtful whether such a general condition makes sense, after all, to formulate
     "consistency" we only need a definiens, and if there were any
     incorrect sentences in the definiens, then generating sentences should probably
     take care of it. }
    if gDefiniens = nil then
    begin
      Consistency := NewInCorFrm; exit;
    end;
    Mizassert(2522,gDefiniens^.nPartialDefinientia.Count <> 0);
    cFrm := NewVerum;
    with gDefiniens^ do
      for i := 0 to nPartialDefinientia.Count - 1 do
        with PartDefPtr(nPartialDefinientia.Items^[i])^ do
      begin
        for j :=  i+1 to nPartialDefinientia.Count - 1 do
        begin
          cFrm2 := NewConj(FrmPtr(nGuard)^.CopyFormula,
                           FrmPtr(PartDefPtr(nPartialDefinientia.Items^[j])^.nGuard)^.CopyFormula);
          case DefSort of
            'm':
              begin
                lFrm1 := FrmPtr(nPartDefiniens)^.CopyFormula;
                lFrm2 := FrmPtr(PartDefPtr(nPartialDefinientia.Items^[j])^.nPartDefiniens)^.CopyFormula
              end;
            'e':
              begin
                lFrm1 := NewEqFrm(NewItTrm,CopyTerm(TrmPtr(nPartDefiniens)));
                lFrm2 := NewEqFrm(NewItTrm,CopyTerm(TrmPtr(PartDefPtr(nPartialDefinientia.Items^[j])^.nPartD
              end;
            else RunTimeError(2505);
          end;
          EqFrm := NewBiCond(lFrm1,lFrm2);
          cFrm := NewConj(cFrm,NewImpl(cFrm2,EqFrm));
        end;
      end;
    if ff in ['M','K'] then
    begin
      WithInFormula(cFrm,ChangeBoundAndIt);
      cFrm := NewUniv(ItTyp^.CopyType,cFrm);
    end;
    Consistency := cFrm;
  end;
```

This code is used in chunk 17b.
Defines:
  Consistency, used in chunk 27c.
Uses ChangeBoundAndIt 15c.

20 ⟨*Existence* 20⟩≡

```
  function Existence(ff: char): FrmPtr;
  var
    lOth,cFrm:FrmPtr;
```

```
    function PartExCond(fFrm: FrmPtr): FrmPtr;
    begin
       fFrm := fFrm^.CopyFormula;
       if fFrm^.FrmSort <> ikError then
       begin
          WithInFormula(fFrm,ChangeBoundAndIt);
          fFrm := NewExis(ItTyp^.CopyType,fFrm);
       end;
       PartExCond := fFrm;
    end;
  var
    z: integer;
  begin
    if gDefiniens = nil then
    begin
       Existence := NewInCorFrm; exit;
    end;
    with gDefiniens^ do
    begin
       if nOtherwise <> nil then lOth := NewVerum;
       cFrm := NewVerum;
       with nPartialDefinientia do
          for z := 0 to Count-1 do
             with PartDefPtr(Items^[z])^ do
          begin
             if gDefiniens^.nOtherwise <> nil then
                lOth := NewConj(lOth,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
             cFrm := NewConj(cFrm,NewImpl(FrmPtr(nGuard)^.CopyFormula,
                                          PartExCond(FrmPtr(nPartDefiniens))));
          end;
       if nOtherwise <> nil then
          cFrm := NewConj(cFrm,NewImpl(lOth,PartExCond(FrmPtr(nOtherwise))));
       Existence := cFrm;
    end;
  end;
```

This code is used in chunk 17b.
Defines:
  Existence, used in chunks 38b and 43d.
Uses ChangeBoundAndIt 15c.

21 ⟨*Parse coherence in equals definition* 21⟩≡

```
  function CoherenceEq: FrmPtr;
  var
    lOth,cFrm: FrmPtr;
    z: integer;
  begin
    if gDefiniens = nil then
    begin CoherenceEq := NewInCorFrm; exit end;
    with gDefiniens^ do
    begin
       mizassert(2598,DefSort = 'e');
       if nOtherwise <> nil then lOth := NewVerum;
       cFrm := NewVerum;
       with nPartialDefinientia do
          for z := 0 to Count-1 do
             with PartDefPtr(Items^[z])^ do
          begin
             if gDefiniens^.nOtherwise <> nil then
                lOth := NewConj(lOth,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
             cFrm := NewConj(cFrm,NewImpl(FrmPtr(nGuard)^.CopyFormula,
                                          NewQualFrm(CopyTerm(TrmPtr(nPartDefiniens)),ItTyp^.CopyType)));
```

```
            end;
         if nOtherwise <> nil then
         begin
            cFrm := NewConj(cFrm,NewImpl(lOth,
                                         NewQualFrm(CopyTerm(TrmPtr(nOtherwise)),ItTyp^.CopyType)));
         end;
         CoherenceEq := cFrm;
      end;
   end;
```

This code is used in chunk 17b.
Defines:
CoherenceEq, used in chunk 38b.

22a ⟨*Change bound variables and iterate (one)* 22a⟩≡

```
procedure ChangeBoundAndIt1(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
  with VarTrmPtr(fTrm)^ do
     case TrmSort of
        ikTrmIt:
           begin
              lTrm := fTrm;
              fTrm := NewVarTrm(ikTrmBound,1);
              dispose(lTrm,Done);
           end;
        ikTrmBound:
           inc(VarNr,2);
     end;
end;
```

This code is used in chunk 17b.
Defines:
ChangeBoundAndIt1, used in chunks 22b and 23a.

22b ⟨*Copy formula with fresh bound variables* 22b⟩≡

```
function NewGuard(fFrm: FrmPtr): FrmPtr;
begin
   if fFrm^.FrmSort <> ikError then
   begin
      fFrm := fFrm^.CopyFormula;
      WithInFormula(fFrm,ChangeBoundAndIt1);
   end;
   NewGuard := fFrm;
end;
```

This code is used in chunk 17b.
Defines:
NewGuard, used in chunk 23b.
Uses ChangeBoundAndIt1 22a.

The only difference with ChangeBoundAndIt1 and this procedure is the ikTrmIt case will call NewVarTrm(ikTrmBound,2) instead of NewVarTrm(ikTrmBound,1). Its significance eludes me at the moment.

22c ⟨*Change bound variables and iterate (two)* 22c⟩≡

```
procedure ChangeBoundAndIt2(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
   with VarTrmPtr(fTrm)^ do
      case TrmSort of
         ikTrmIt:
            begin
               lTrm := fTrm;
```

```
                fTrm := NewVarTrm(ikTrmBound,2);
                dispose(lTrm,Done);
             end;
          ikTrmBound: inc(VarNr,2);
       end;
  end;
```

This code is used in chunk 17b.
Defines:
ChangeBoundAndIt2, used in chunk 23a.

23a ⟨*Partial uniqueness condition* 23a⟩≡

```
  function PartUniCond(fFrm: FrmPtr): FrmPtr;
  var
     cFrm1,cFrm2: FrmPtr;
  begin
     if fFrm^.FrmSort <> ikError then
     begin
        cFrm1 := fFrm^.CopyFormula;
        cFrm2 := fFrm^.CopyFormula;
        WithInFormula(cFrm1,ChangeBoundAndIt1);
        WithInFormula(cFrm2,ChangeBoundAndIt2);
        fFrm := NewImpl(NewConj(cFrm1,cFrm2),
                        NewEqFrm(NewVarTrm(ikTrmBound,1),NewVarTrm(ikTrmBound,2)));
     end;
     PartUniCond := fFrm;
  end;
```

This code is used in chunk 17b.
Defines:
PartUniCond, used in chunk 23b.
Uses ChangeBoundAndIt1 22a and ChangeBoundAndIt2 22c.

23b ⟨*Parse uniqueness condition* 23b⟩≡

```
  function Uniqueness: FrmPtr;
  var
     cFrm,lOth: FrmPtr;
     lTyp: TypPtr;
     z: integer;
  begin
     if gDefiniens = nil then begin Uniqueness := NewInCorFrm; exit end;
     with gDefiniens^ do
     begin
        if nOtherwise <> nil then lOth := NewVerum;
        cFrm := NewVerum;
        with nPartialDefinientia do
           for z := 0 to Count-1 do
              with PartDefPtr(Items^[z])^ do
           begin
              if gDefiniens^.nOtherwise <> nil then
                 lOth := NewConj(lOth,NewNegDis(NewGuard(FrmPtr(nGuard))));
              cFrm := NewConj(cFrm,NewImpl(NewGuard(FrmPtr(nGuard)),
                                           PartUniCond(FrmPtr(nPartDefiniens))));
           end;
        if nOtherwise <> nil then
           cFrm := NewConj(cFrm,NewImpl(lOth,PartUniCond(FrmPtr(nOtherwise))));
     end;
     lTyp := ItTyp^.CopyType;
     Uniqueness := NewUniv(lTyp^.CopyType,NewUniv(lTyp,cFrm));
  end;
```

This code is used in chunk 17b.
Defines:
Uniqueness, used in chunk 38b.
Uses NewGuard 22b and PartUniCond 23a.

We need to forward declare Justify when parsing the correctness conditions.

24 ⟨*Parse correctness conditions* 24⟩≡

```
procedure Justify(ThesisId, fLabId: integer; fThesis: FrmPtr); forward;

procedure Correctness;
var
   cFrm: FrmPtr;
   k,lCorrCondNr: integer;
begin
   while InFile.Current.Kind='Y' do
   begin
      lCorrCondNr := InFile.Current.Nr;
      InFile.InPos(CurPos);
      mizassert(2514,gCorrCond[lCorrCondNr]<>nil);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(Nr2CorrEl[ lCorrCondNr]);
      {$ENDIF}
      Justify(0,0,gCorrCond[lCorrCondNr]);
      if lCorrCondNr <> 0 then
      begin
         dispose(gCorrCond[lCorrCondNr],Done);
         gCorrCond[lCorrCondNr] := nil;
      end;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(Nr2CorrEl[ lCorrCondNr]);
      {$ENDIF}
   end;
   if InFile.Current.Kind= ikItmCorrectness then
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elCorrectness);
      {$ENDIF}
      InFile.InPos(CurPos);
      cFrm := NewVerum;
      for k := 1 to CorrCondNbr do
         if gCorrCond[k] <> nil then
         begin
            cFrm := NewConj(cFrm,gCorrCond[k]);
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart0(Nr2CorrEl[ k]);
            AReport.Out_Formula(gCorrCond[k]);
            AReport.Out_XElEnd(Nr2CorrEl[ k]);
            {$ENDIF}
         end;
      Justify(0,0,cFrm);
      dispose(cFrm,Done);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elCorrectness);
      {$ENDIF}
   end;
end;
```

Root chunk (not used in this document).
Defines:
Correctness, used in chunks 59, 121, and 129.
Uses Justify 96b.

1.3 Checking Loci

Loci occur in definitions and registrations. We need to check they are accessible and defined.

25 ⟨*Analyzer methods* 5a⟩+≡
⟨*Check Loci constants* 26a⟩
⟨*Check loci constants in definiens* 26b⟩
⟨*Analyze sentence* 27a⟩
⟨*New in correlate definition* 27b⟩
⟨*Read definiens* 27c⟩
⟨*Constructor for* `DefNode` 29a⟩
⟨*Write definiens* 29b⟩
⟨*Read pattern* 30b⟩
⟨*Determine abstract notation number* 31a⟩
⟨*Get pattern* 31b⟩
⟨*Initialize access* 31c⟩
⟨*Check access* 31d⟩
⟨*Initialize loci for cluster* 32a⟩
⟨*Check all loci are accessible in type* 32b⟩
⟨*Check all loci are accessible in term* 33a⟩
⟨*Definition predicate pattern* 33b⟩
⟨*Create list of constant terms* 34a⟩
⟨*Create a list of terms* 34b⟩
⟨*Redefine predicate pattern* 34c⟩
⟨*Notation predicate pattern* 35⟩
⟨*Insert predicate* 37a⟩
⟨*Parse definition of predicate — tail* 37b⟩
⟨*Parse definition of attribute — tail* 37c⟩
⟨*Parse specification* 38a⟩
⟨*Parse functor definition pattern* 38b⟩
⟨*Parse redefinition specification* 39a⟩
⟨*Parse pattern in functor redefinition* 39b⟩
⟨*Parse notation in functor pattern* 41⟩
⟨*Insert functor* 43a⟩
⟨*Parse definition of functor — tail* 43b⟩
⟨*Create list of loci* 43c⟩
⟨*Parse mode pattern in definition* 43d⟩
⟨*Parse mode pattern in redefinition* 44⟩
⟨*Parse mode pattern for notation* 45⟩
⟨*Insert a mode* 47a⟩
⟨*Parse definition of an expandable mode* 47b⟩
⟨*Parse predicate or attribute pattern in definition* 47c⟩
⟨*Parse pattern for predicate or attribute redefinition* 48⟩
⟨*Parse notation in a predicate or attribute pattern* 49⟩
⟨*Insert predicate or attribute* 50⟩
⟨*Analyze cluster* 51a⟩
⟨*Add items to a cluster* 51b⟩
⟨*Define existential cluster* 52⟩
⟨*Define conditional cluster* 53⟩
⟨*Define functorial cluster* 54⟩
⟨*Collect loci* 56a⟩
⟨*Find pattern* 56b⟩
⟨*Define reduction* 58⟩
⟨*Define identify* 59⟩
⟨*Define property* 62a⟩
⟨*Parse definition of mode — tail* 62b⟩
⟨*Set structure* 63a⟩
⟨*Analyze selector* 63b⟩
⟨*Define a structure* 64⟩
⟨*Parse a reservation* 70⟩
⟨*Spread local predicates* 71⟩
⟨*Decompose formula* 72a⟩
⟨*Spread atomic formula* 72b⟩
⟨*Chopping definientia(?)* 73⟩

⟨*Change bound variable to declaration(?)* 75a⟩
⟨*Mark term as taken* 75b⟩
⟨*Chop variables* 75c⟩
⟨*Is* `Position` *in the collection?* 77a⟩
⟨*Chop conclusion* 77b⟩
⟨*Dispose level* 78b⟩
⟨*Reasoning* 78c⟩
⟨*Per cases reasoning* 82⟩
⟨*Demonstration* 85⟩
⟨*Change declared constant to bound variable* 87a⟩
⟨*Skeletonize list* 87b⟩
⟨*Skeletonize sentence* 88a⟩
⟨*Diffuse Reasoning* 88b⟩
⟨*New list of universally quantified variables* 90⟩
⟨*Change skeletonized fixed variable to bound variable* 91a⟩
⟨*New universal list (one)* 91b⟩
⟨*Reasoning result* 91c⟩
⟨*Diffuse per cases reasoning* 92⟩
⟨*Diffuse statement* 95⟩
⟨*Hereby* 96a⟩
⟨*Justify* 96b⟩
⟨*Regular statement* 97⟩
⟨*Parse a statement* 100⟩

This code is used in chunk 3c.

26a ⟨*Check Loci constants* 26a⟩≡

```
procedure CheckLocConst(var fTrm: TrmPtr);
begin
   with VarTrmPtr(fTrm)^ do
      if TrmSort=ikTrmConstant then
         if (FixedVar[VarNr].nSkelConstNr=0) and (VarNr>g.DemBase) then
            gConstErr := true;
end;
```

This code is used in chunk 25.
Defines:
`CheckLocConts`, never used.
Uses `gConstErr` 17a.

26b ⟨*Check loci constants in definiens* 26b⟩≡

```
procedure CheckLocConstInDefiniens(fDef: DefPtr);
var
   z: integer;
begin
   with fDef^ do
   begin
      with nPartialDefinientia do
         for z := 0 to Count-1 do
            with PartDefPtr(Items^[z])^ do
         begin
            case DefSort of
               'm': WithInFormula(FrmPtr(nPartDefiniens),CheckLocConst);
               'e': WithInTerm(TrmPtr(nPartDefiniens),CheckLocConst);
               else RunTimeError(2506);
            end;
            WithInFormula(FrmPtr(nGuard),CheckLocConst);
         end;
      if nOtherWise<>nil then
         case DefSort of
            'm': WithInFormula(FrmPtr(nOtherWise),CheckLocConst);
            'e': WithInTerm(TrmPtr(nOtherWise),CheckLocConst);
            else RunTimeError(2507);
```

```
            end;
      end;
   end;
```

This code is used in chunk 25.
Defines:
CheckLocConstInDefiniens, used in chunk 27c.

27a ⟨*Analyze sentence* 27a⟩≡

```
{ Uzyc w ReadSentence !!!} { Used in ReadSentence !!!}
function AnalyzeSnt(fSnt: ExpPtr; fNeg: boolean): FrmPtr;
var
   lFrm: FrmPtr;
begin
   BoundVarNbr := 0;
   lFrm := fSnt^.Analyze;
   dispose(fSnt,Done);
   if fNeg then lFrm := NewNegDis(lFrm);
   AnalyzeSnt := lFrm;
end;
```

This code is used in chunk 25.
Defines:
AnalyzeSnt, used in chunk 27c.
Uses Analyze 138 and ReadSentence 7a.

27b ⟨*New in correlate definition* 27b⟩≡

```
function NewInCorDef: DefPtr;
var
   lColl: MCollection;
begin
   lColl.Init(0,0);
   NewInCorDef := new(DefPtr,Init(ikError,lColl,NewInCorFrm));
end;
```

This code is used in chunk 25.
Defines:
NewInCorDef, used in chunk 27c.

27c ⟨*Read definiens* 27c⟩≡

```
procedure ReadDefiniens(Negate: boolean; const aPrim: MList; fType: TypPtr);
var
   k,lLabId: integer;
   lPartDef: PObject;
   lGuard: FrmPtr;
   lPartialPart: MCollection;
   lOtherwise: PObject;
   pDefiniens: DefObj;
   lPartDefPtr: PartDefPtr;
   lPartDefiniens,llGuard: ExpPtr;
   z: integer;
begin
   gDefNode.MeansOccurs := ' ';
   gDefNode.Specified := false;
   gDefiniens := nil;
   if InFile.Current.Kind in ['m','e'] then
   begin
      gDefNode.MeansOccurs := InFile.Current.Kind;
      InFile.InWord;
      InFile.InInt(lLabId);
      gDefNode.Pos1 := CurPos;
      gDefNode.fPrimaries.Init(gDefNode.Length,1);
      for k := 1 to gDefNode.Length do
      begin
```

```
      gDefNode.fPrimaries.Insert(TypPtr(aPrim.Items^[k-1])^.CopyType);
   end;
   if fType <> nil then
   begin
      gDefNode.Specified := true;
      fType := AdjustedType(fType);
      fType^.WithinType(ChangeToLoci);
      gDefNode.fPrimaries.Insert(fType);
      { Ten typ nalezy dysponowac !
        This type is a must-have! }
   end;
   gDefNode.LabNr := InFile.Current.Nr;
   gDefNode.LabId :=  lLabId;
   InFile.InPos(gDefNode.Pos2);
   with pDefiniens do
   begin
      InFile.InWord; DefSort := InFile.Current.Kind;
      nPartialDefinientia.Init(2,2);
      InFile.InWord;
      while InFile.Current.Kind <> ';' do
      begin
         case DefSort of
            'm': lPartDefiniens := LoadFormula;
            'e': lPartDefiniens := LoadTerm;
            else RunTimeError(2508);
         end;
         llGuard := LoadFormula;
         lPartDefPtr := new(PartDefPtr,Init(lPartDefiniens,llGuard));
         nPartialDefinientia.Insert(lPartDefPtr);
         InFile.InWord;
      end;
      InFile.InWord;
      case InFile.Current.Kind of
         'n': nOtherwise := nil;
         'o':
            case DefSort of
               'm': nOtherwise := LoadFormula;
               'e': nOtherwise := LoadTerm;
               else RunTimeError(2509);
            end;
         else RunTimeError(2520);
      end;
   end;
   lPartialPart.Init(0,4);
   with pDefiniens,nPartialDefinientia do
   begin
      for z := 0 to Count-1 do
         with PartDefPtr(Items^[z])^ do
      begin
         case DefSort of
            'm': lPartDef := AnalyzeSnt(ExpPtr(nPartDefiniens),Negate);
            'e': lPartDef := AnalyzeTerm(ExpPtr(nPartDefiniens));
            else RunTimeError(2519);
         end;
         lGuard := AnalyzeSnt(ExpPtr(nGuard),false);
         lPartialPart.Insert(new(PartDefPtr,Init(lPartDef,lGuard)));
      end;
      lOtherwise := nil;
      if nOtherwise <> nil then
         case DefSort of
```

```
                'm': lOtherwise := AnalyzeSnt(ExpPtr(nOtherWise),Negate);
                'e': lOtherwise := AnalyzeTerm(ExpPtr(nOtherWise));
                else RunTimeError(2510);
              end;
        end;
        gDefiniens := new(DefPtr,Init(gDefNode.MeansOccurs,lPartialPart,lOtherwise));
        { kontrola stalych lokalnych }
        { local constant control }
        gConstErr := false;
        CheckLocConstInDefiniens(gDefiniens);
        if gConstErr then
        begin ErrImm(69); gDefiniens := NewInCorDef end;
        if gDefiniens^.nPartialDefinientia.Count <> 0 then
           gCorrCond[ord(syConsistency)] := Consistency(gDefNode.Kind);
        if gRedef then gCorrCond[ord(syCompatibility)] := Compatibility(gDefNode.Kind);
        InFile.InWord;
     end;
  end;
```

This code is used in chunk 25.
Defines:
ReadDefiniens, used in chunks 33–35, 38b, 39b, 41, 43–45, and 47–49.
Uses AnalyzeSnt 27a, AnalyzeTerm 7c, CheckLocConstInDefiniens 26b, Compatibility 18b, Consistency 19, gConstErr 17a, gDefNode 17a, and NewInCorDef 27b.

29a ⟨*Constructor for* DefNode 29a⟩≡

```
  var DefinitionList: MCollection;
      gEssentials: IntSequence;

  constructor DefNode.Init(fMeansOccurs,fKind: char; fLab,fLabId: integer;
                           fDef: DefPtr;
                           fEntry: RSNENTRY);
  begin
     nMeansOccurs := fMeansOccurs;
     nConstructor.Kind := gDefNode.Kind; nConstructor.Nr := gWhichOne;
     SkId := fLab; SkLabId :=  fLabId; DDef := fDef;
     SkVarNbr := g.VarNbr;
     SkIt := g.GenCount;
     case fKind of
        'R','V':;
        else inc(SkIt)
     end;
     nPrefix := fEntry;
     nEssentials.CopySequence(gEssentials);
     move(gDefNode.fPrimaries,nPrimaryList,SizeOf(gDefNode.fPrimaries));
  end;
```

This code is used in chunk 25.
Defines:
DefinitionList, used in chunks 29b, 117c, 119, 121, 129, and 132.
DefNode.Init, never used.
gEssentials, used in chunk 29b.
Uses DefNode 4b and gDefNode 17a.

29b ⟨*Write definiens* 29b⟩≡

```
  procedure WriteDefiniens;
  var
     k,r: integer;
  begin
     if gDefNode.MeansOccurs <> ' ' then
     begin
        gEssentials.Init(0);
        for k := gSuperfluous+1 to gDefNode.Length do
```

```
            r := gEssentials.Insert(k);
         if gDefNode.Specified then
            r := gEssentials.Insert(gDefNode.Length+1);
         DefinitionList.Insert(new(DefNodePtr,
                                   Init(gDefNode.MeansOccurs,gDefNode.Kind,gDefNode.LabNr,gDefNode.LabId,gDefiniens,
      end;
   end;
```

This code is used in chunk 25.
Defines:
`WriteDefiniens`, used in chunk 121.
Uses `DefinitionList` 29a, `DefNodePtr` 4b, `gDefNode` 17a, and `gEssentials` 29a.

30a ⟨*Read visible* 30a⟩≡

```
var ConstrError: boolean;
function ReadVisible: integer;
var
   lInt: integer;
begin
   if InFile.Current.Kind = ';' then
   begin
      ReadVisible := -1;
      exit;
   end;
   if InFile.Current.Nr<>0 then
   begin
      lInt := FixedVar[InFile.Current.Nr].nSkelConstNr;
      LociOcc[lInt] := true
   end
   else begin
      lInt := 0;
      ConstrError := true
   end;
   ReadVisible := lInt;
   InFile.InWord;
end;
```

Root chunk (not used in this document).
Defines:
`ConstrError`, never used.
`ReadVisible`, never used.

30b ⟨*Read pattern* 30b⟩≡

```
procedure ReadPattern(var aFormNr: integer; var aVisible: IntSequence);
var
   r: integer;
begin
   { Przewiniecie formatu konstruktora }
   { Scroll the constructor format }
   aFormNr := InFile.Current.Nr;
   InFile.InPos(CurPos);
   aVisible.Init(0);
   InFile.InWord;
   while InFile.Current.Kind <> ';' do
   begin
      r := aVisible.Insert(InFile.Current.Nr);
      InFile.InWord;
   end;
end;
```

This code is used in chunk 25.
Defines:
`ReadPattern`, used in chunks 31b and 56b.

31a ⟨*Determine abstract notation number* 31a⟩≡
```
function AbsNotatNr(nk: NotationKind): integer;
begin
  AbsNotatNr :=  1 + Notat[nk].Count +  Notat[nk].fExtCount - NotatBase[nk];
end;
```
This code is used in chunk 25.
Defines:
AbsNotatNr, used in chunks 31b and 64.

31b ⟨*Get pattern* 31b⟩≡
```
procedure GetPattern(aKind: NotationKind; var aPattern: PatternPtr);
var
   k: integer;
begin
   aPattern := new(PatternPtr,Init(aKind, AbsNotatNr(aKind), ArticleID));
   with aPattern^ do
   begin
      ReadPattern(fFormNr,Visible);
      { Inicjalizacja } { Initialization }
      fPrimTypes.Init(dPrimLength);
      for k := 1 to dPrimLength do
         fPrimTypes.Insert(gPrimaries[k]^.CopyType);
      { Wyliczenie i zamarkowanie listy visible }
      { Enumerate and mark the visible list }
      for k := 0 to Visible.fCount-1 do
         if Visible.fList^[k] <> 0 then
            Visible.fList^[k] := FixedVar[Visible.fList^[k]].nSkelConstNr
         else begin
            Visible.fList^[k] := 0;
            fFormNr := 0
         end;
   end;
end;
```
This code is used in chunk 25.
Defines:
GetPattern, used in chunks 33–35, 38b, 39b, 41, 43–45, 47–49, and 64.
Uses AbsNotatNr 31a and ReadPattern 30b.

31c ⟨*Initialize access* 31c⟩≡
```
procedure InitAccess;
begin
   FillChar(LociOcc,SizeOf(LociOcc),0);
end;
```
This code is used in chunk 25.
Defines:
InitAccess, used in chunks 31d, 52–54, 58, 59, 62a, and 64.

The loci variables need to be accessible in the pattern of a definition.

31d ⟨*Check access* 31d⟩≡
```
procedure CheckAccess(aPattern: PatternPtr);
var
   i,k: integer;
begin
   InitAccess;
   with aPattern^ do
   begin
      { Kontrola poprawnosci konstruktora :
        - czy ma poprawny typ
        - czy kazdy lokus jest dostepny
      Constructor validation:
        - whether it has the correct type
```

```
          - whether each locus is available }
        if fFormNr<>0 then
        begin
           { Zamarkowanie listy visible } { Mark the visible list }
           for k := 0 to Visible.fCount-1 do
              if Visible.fList^[k] <> 0 then
                 LociOcc[ord(Visible.fList^[k])] := true;
           for i := fPrimTypes.Count-1 downto 0 do
           begin
              if TypPtr(fPrimTypes.Items^[i])^.TypSort=ikError
              then begin fFormNr := 0; exit end;
              if not LociOcc[i+1] then
              begin fFormNr := 0; ErrImm(100); exit end;
              TypPtr(fPrimTypes.Items^[i])^.WithinType(SetLociOcc);
           end;
        end;
     end;
  end;
```

This code is used in chunk 25.
Defines:
  `CheckAccess`, used in chunks 33–35, 38b, 39b, 41, 43–45, 47–49, and 64.
Uses `InitAccess` 31c and `SetLociOcc` 5c.

32a ⟨*Initialize loci for cluster* 32a⟩≡

```
  procedure InitLociForCluster(aClusterPtr: AttrCollectionPtr);
  begin
     aClusterPtr := CopyCluster(aClusterPtr);
     aClusterPtr^.WithinAttrCollection(ChangeToLoci);
     aClusterPtr^.WithinAttrCollection(SetLociOcc);
     dispose(aClusterPtr,Done);
  end;
```

This code is used in chunk 25.
Defines:
  `InitLociForCluster`, used in chunks 52 and 53.
Uses `SetLociOcc` 5c.

32b ⟨*Check all loci are accessible in type* 32b⟩≡

```
  function AllLociAccessibleInTyp(const aTypList: MCollection; aTyp:TypPtr): boolean;
  var
     i: integer;
  begin
     AllLociAccessibleInTyp := false;
     aTyp := aTyp^.CopyType;
     aTyp^.WithinType(ChangeToLoci);
     aTyp^.WithinType(SetLociOcc);
     for i := aTypList.Count-1 downto 0 do
     begin
        if TypPtr(aTypList.Items^[i])^.TypSort=ikError  then
        begin Dispose(aTyp,Done); exit end;
        if not LociOcc[i+1] then
        begin Dispose(aTyp,Done); exit end;
        TypPtr(aTypList.Items^[i])^.WithinType(SetLociOcc);
     end;
     AllLociAccessibleInTyp := true;
     Dispose(aTyp,Done);
  end;
```

This code is used in chunk 25.
Defines:
  `AllLociAccessibleInTyp`, used in chunks 52, 53, and 62a.
Uses `SetLociOcc` 5c.

33a ⟨*Check all loci are accessible in term* 33a⟩≡

```
function AllLociAccessibleInTrm(const aTypList: MCollection; aTrm:TrmPtr): boolean;
var
   i: integer;
begin
   AllLociAccessibleInTrm := false;
   aTrm := CopyTerm(aTrm);
   WithinTerm(aTrm,ChangeToLoci);
   WithinTerm(aTrm,SetLociOcc);
   for i := aTypList.Count-1 downto 0 do
   begin
      if TypPtr(aTypList.Items^[i])^.TypSort=ikError then
      begin
         Dispose(aTrm,Done);
         exit;
      end;
      if not LociOcc[i+1] then
      begin
         Dispose(aTrm,Done);
         exit;
      end;
      TypPtr(aTypList.Items^[i])^.WithinType(SetLociOcc);
   end;
   AllLociAccessibleInTrm := true;
   DisposeTrm(aTrm);
end;
```

This code is used in chunk 25.
Defines:
`AllLociAccessibleInTrm`, used in chunks 54 and 58.
Uses `SetLociOcc` 5c.

33b ⟨*Definition predicate pattern* 33b⟩≡

```
procedure DefPredPattern;
var
   lPattern: PatternPtr;
begin
   InFile.InPos(CurPos);
   InFile.InWord;
   gDefPos := CurPos;
   GetPattern(noPredicate,lPattern);
   CheckAccess(lPattern);
   Notat[noPredicate].InsertExt(lPattern);
   InFile.InWord;
   with Notat[noPredicate] do
      PatternPtr(Items^[Count+fExtCount-1])^.rConstr.Kind := ikFrmPred;
   gProperties.Properties := [];
   gProperties.nFirstArg := 0;
   gProperties.nSecondArg := 0;
   RedefAntonym := false;
   gDefNode.Kind := 'R';
   with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
   begin
      gDefNode.Length := fPrimTypes.Count;
      ReadDefiniens(false,fPrimTypes,nil);
   end;
end;
```

This code is used in chunk 25.
Defines:
`DefPredPattern`, used in chunk 121.
Uses `CheckAccess` 31d, `gDefNode` 17a, `GetPattern` 31b, and `ReadDefiniens` 27c.

34a ⟨*Create list of constant terms* 34a⟩≡

```
function CreateConstList(const aList: IntSequence): TrmList;
var
   lTrmList: TrmList;
   k: integer;
begin
   lTrmList := nil;
   for k := aList.fCount-1 downto 0 do
      lTrmList := NewTrmList(NewVarTrm(ikTrmConstant,ConstNr[aList.fList^[k]]),lTrmList);
   CreateConstList := lTrmList;
end;
```

This code is used in chunk 25.
Defines:
CreateConstList, used in chunks 34c, 35, 39b, 41, 44, 45, 48, and 49.

34b ⟨*Create a list of terms* 34b⟩≡

```
function CreateTrmList(const aList: IntSequence): TrmList;
var
   lTrmList: TrmList;
   k: integer;
begin
   lTrmList := nil;
   for k := aList.fCount-1 downto 0 do
   begin
      if aList.fList^[k]=0 then
      begin
         lTrmList := InCorrTrmList;
         break;
      end;
      lTrmList := NewTrmList(NewVarTrm(ikTrmConstant,aList.fList^[k]),lTrmList);
   end;
   CreateTrmList := lTrmList;
end;
```

This code is used in chunk 25.
Defines:
CreateTrmList, used in chunk 56b.

34c ⟨*Redefine predicate pattern* 34c⟩≡

```
procedure RedefPredPattern;
var
   K:integer;
   lArgs: TrmList;
   lPattern: PatternPtr;
label Found;
begin
   gRedef := true;
   InFile.InPos(CurPos);
   InFile.InWord;
   gDefPos := CurPos;
   GetPattern(noPredicate, lPattern);
   CheckAccess(lPattern);
   Notat[noPredicate].InsertExt(lPattern);
   InFile.InWord;
   gProperties.Properties := [];
   gProperties.nFirstArg := 0;
   gProperties.nSecondArg := 0;
   RedefAntonym := false;
   fillchar(gSubstTrm,sizeof(gSubstTrm),0);
   with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
   begin
      rConstr.Kind := ikFrmPred;
```

```
        if fFormNr<>0 then
        begin
           lArgs := CreateConstList(Visible);
           for K := Count-1 downto 0 do
              if (OriginalNr(coPredicate,PatternPtr(Items^[k])^.rConstr.Nr)=0) and
                    (PatternPtr(Items^[k])^.fFormNr = fFormNr) and
                    CheckTypes(Items^[k], lArgs)
              then
              begin
                 if PatternPtr(Items^[k])^.fAntonymic then RedefAntonym := true;
                 if CompatibleArgs(PatternPtr(Items^[k])^.fPrimTypes.Count) then
                 begin
                    gWhichOne := PatternPtr(Items^[k])^.rConstr.Nr;
                    with ConstrPtr(Constr[coPredicate].Items^[ gWhichOne])^ do
                    begin
                       gSuperfluous := dPrimLength- nPrimaries.Count;
                       GetProperties(gProperties);
                    end;
                    with gProperties do
                    begin
                       inc(nFirstArg,gSuperfluous);
                       inc(nSecondArg,gSuperfluous);
                    end;
                 end;
                 goto Found;
              end;
           ErrImm(112);
           Found:
              DisposeListOfTerms(lArgs);
        end;
     end;
     DisposeSubstTrm;
     gDefNode.Kind := 'R';
     with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
     begin
        gDefNode.Length := fPrimTypes.Count;
        ReadDefiniens(RedefAntonym,fPrimTypes,nil);
     end;
  end;
```

This code is used in chunk 25.
Defines:
RedefPredPattern, used in chunk 121.
Uses CheckAccess 31d, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

35 ⟨*Notation predicate pattern* 35⟩≡

```
  // ###TODO: potential BUG here - take care of setting fKind of Patterns
  //          properly - it is not clear here
  procedure NotatPredPattern;
  var
     K:integer;
     lArgs: TrmList;
     lPattern, origin: PatternPtr;
     lSynonym: boolean;
  label Found1,Found2;
  begin
     gRedef := true;
     InFile.InPos(CurPos); InFile.InWord;
     gDefPos := CurPos;
     GetPattern(noPredicate, lPattern);
     CheckAccess(lPattern);
     Notat[noPredicate].InsertExt(lPattern);
```

```
InFile.InWord;
lSynonym := InFile.Current.Kind=ikMscSynonym;
InFile.InPos(CurPos); InFile.InWord;
GetPattern(noPredicate, origin);
CheckAccess(origin);
InFile.InWord;
gProperties.Properties := [];
gProperties.nFirstArg := 0;
gProperties.nSecondArg := 0;
RedefAntonym := false;
fillchar(gSubstTrm,sizeof(gSubstTrm),0);
with Notat[noPredicate], origin^ do
begin
   rConstr.Kind := ikFrmPred;
   if fFormNr<>0 then
   begin
      lArgs := CreateConstList(Visible);
      for K := Count-1 downto 0 do
         if (OriginalNr(coPredicate,PatternPtr(Items^[k])^.rConstr.Nr)=0) and
               (PatternPtr(Items^[k])^.fFormNr=fFormNr) and
               CheckTypes(Items^[k],lArgs)
         then
         begin
            if PatternPtr(Items^[k])^.fAntonymic then RedefAntonym := true;
            if CompatibleArgs(PatternPtr(Items^[k])^.fPrimTypes.Count) then
            begin
               gWhichOne := OriginalNr(coPredicate,PatternPtr(Items^[k])^.rConstr.Nr);
               if gWhichOne = 0 then
                  gWhichOne := PatternPtr(Items^[k])^.rConstr.Nr;
               with ConstrPtr(Constr[coPredicate].Items^[ gWhichOne])^ do
               begin
                  gSuperfluous := dPrimLength- nPrimaries.Count;
                  GetProperties(gProperties);
               end;
               with gProperties do
               begin
                  inc(nFirstArg,gSuperfluous);
                  inc(nSecondArg,gSuperfluous);
               end;
            end;
            goto Found2;
         end;
      ErrImm(112);
      Found2:
         DisposeListOfTerms(lArgs);
   end;
end;
with lPattern^, rConstr do
begin
   Kind := ikFrmPred;
   Nr := gWhichOne;
   fAntonymic := lSynonym = RedefAntonym;
   fRedefNr :=  K+1;
end;
DisposeSubstTrm;
gDefNode.Kind := 'R';
// with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
//  GB: To ponizej jest chyba zbyteczne bo origin nie jest teraz dopisywany do Notat[noPredicate]
{GB: This below is probably redundant because origin is now not added to Notat[noPredicate]}
with origin^ do
```

```
    begin
      gDefNode.Length := fPrimTypes.Count;
      ReadDefiniens(RedefAntonym,fPrimTypes,nil);
    end;
  end;
```

This code is used in chunk 25.
Defines:
NotatPredPattern, used in chunk 132.
Uses CheckAccess 31d, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

37a ⟨*Insert predicate* 37a⟩≡

```
  procedure InsertPredicate;
  var
    lConstr: ConstrPtr;
    lAbsNr: integer;
  begin
    lAbsNr :=  1 + Constr[coPredicate].Count - ConstrBase[coPredicate];
    with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
      lConstr := new(ConstrPtr,
                     InitForPattern(coPredicate,lAbsNr,ArticleID,fPrimTypes));
    lConstr^.SetProperties(gProperties);
    lConstr^.SetRedef(gWhichOne, gSuperfluous);
    gWhichOne :=  Constr[coPredicate].Count;
    Constr[coPredicate].Insert(lConstr);
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_Constructor(lConstr, gWhichOne);
    {$ENDIF}
  end;
```

This code is used in chunk 25.
Defines:
InsertPredicate, used in chunk 121.

37b ⟨*Parse definition of predicate — tail* 37b⟩≡

```
  procedure DefPredTail;
  begin
    with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
    begin
      rConstr.Nr := gWhichOne;
      fAntonymic := RedefAntonym;
    end;
  end;
```

This code is used in chunk 25.
Defines:
DefPredTail, used in chunk 121.

37c ⟨*Parse definition of attribute — tail* 37c⟩≡

```
  procedure DefAttrTail;
  begin
    with Notat[noAttribute], PatternPtr(Items^[Count+fExtCount-1])^ do
    begin
      rConstr.Nr := gWhichOne;
      fAntonymic := RedefAntonym;
    end;
  end;
```

This code is used in chunk 25.
Defines:
DefAttrTail, used in chunk 121.

38a ⟨*Parse specification* 38a⟩≡

```
procedure Specification;
begin
   gSpecified := InFile.Current.Kind=ikMscSpecification;
   if InFile.Current.Kind = ikMscSpecification then
   begin
      gFraenkelTermAllowed := false;
      ItTyp := ReadType;
      gFraenkelTermAllowed := true;
      Infile.InWord
   end
   else ItTyp := AnyTyp^.CopyType;
end;
```

This code is used in chunk 25.
Defines:
`Specification`, used in chunks 38b, 39a, and 43d.
Uses `ReadType` 7b.

38b ⟨*Parse functor definition pattern* 38b⟩≡

```
procedure DefFuncPattern;
var
   lPattern: PatternPtr;
begin
   InFile.InPos(CurPos);
   InFile.InWord;
   gDefPos := CurPos;
   GetPattern(noFunctor, lPattern);
   CheckAccess(lPattern);
   Notat[noFunctor].InsertExt(lPattern);
   InFile.InWord;
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                            rConstr.Kind := ikTrmFunctor;
   Specification;
   { InitForPattern tworzy kopie typu. }
   { InitForPattern creates copies of the type.}
   { Definiens tworzy kopie typu (AdjustedType), po wyrzuceniu optymalizacji
     na "Any".
   }
   { Definiens creates copies of the type (AdjustedType) after rolling
     the optimization to "Any". }
   gProperties.Properties := [];
   gProperties.nFirstArg := 0;
   gProperties.nSecondArg := 0;
   gDefNode.Kind := 'K';
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
   begin
      gDefNode.Length := fPrimTypes.Count;
      ReadDefiniens(false,fPrimTypes,ItTyp);
   end;
   if gDefNode.MeansOccurs = 'e' then
      gCorrCond[ord(syCoherence)] := CoherenceEq
   else
   begin
      gCorrCond[ord(syExistence)] := Existence('K');
      gCorrCond[ord(syUniqueness)] := Uniqueness;
   end;
end;
```

This code is used in chunk 25.
Defines:
`DefFuncPattern`, used in chunk 121.

Uses CheckAccess 31d, CoherenceEq 21, Existence 20, gDefNode 17a, GetPattern 31b, ReadDefiniens 27c, Specification 38a, and Uniqueness 23b.

39a ⟨*Parse redefinition specification* 39a⟩≡

```
procedure RedefSpecification(fTyp: TypPtr; Err: integer);
var
   lTyp: TypPtr;
begin
   if gWhichOne = 0 then
   begin
      Specification;
      exit;
   end;
   { Jezeli nie udalo sie zidentyfikowac redefiniowany fuktor,
     to traktujemy to jako definicje nowego funktora
   }
   { If we failed to identify the redefined functor,
     we treat it as the definition of a new functor }
   gSpecified := InFile.Current.Kind=ikMscSpecification;
   if InFile.Current.Kind=ikMscSpecification then
   begin
      ItTyp := ReadType;
      Infile.InWord;
      { co sie dzieje jezeli fTyp jest niepoprawny ? }
      { Nic sie nie rozszerza do niepoprawnego.}
      { what happens if fType is incorrect? }
      {Nothing expands to invalid.}
      if ItTyp^.TypSort<>ikError then
      begin
         lTyp := fTyp^.CopyType;
         lTyp^.WithinType(ChangeToConst);
         if not lTyp^.IsWiderThan(ItTyp^.CopyType) then ErrImm(Err);
         dispose(lTyp,Done);
      end;
      exit;
   end;
   { Jezeli specyfikacja jest opuszczona to jest to typ oryginalu. }
   { If the specification is omitted, this is the original type. }
   ItTyp := fTyp^.CopyType;
   ItTyp^.WithinType(ChangeToConst);
end;
```

This code is used in chunk 25.
Uses ChangeToConst 6a, ReadType 7b, and Specification 38a.

39b ⟨*Parse pattern in functor redefinition* 39b⟩≡

```
procedure RedefFuncPattern;
var
   K: integer;
   lArgs: TrmList;
   lPattern: PatternPtr;
label Found;
begin
   gRedef := true;
   InFile.InPos(CurPos); InFile.InWord;
   gDefPos := CurPos;
   GetPattern(noFunctor, lPattern);
   CheckAccess(lPattern);
   Notat[noFunctor].InsertExt(lPattern);
   InFile.InWord;
   fillchar(gSubstTrm,sizeof(gSubstTrm),0);
   gProperties.Properties := [];
```

```
gProperties.nFirstArg := 0;
gProperties.nSecondArg := 0;
with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
begin
   rConstr.Kind := ikTrmFunctor;
   if fFormNr<>0 then
   begin
      lArgs := CreateConstList(Visible);
      for K := Count-1 downto 0 do
         if (OriginalNr(coFunctor,PatternPtr(Items^[K])^.rConstr.Nr)=0) and
               (PatternPtr(Items^[K])^.fFormNr=fFormNr) and
               CheckTypes(Items^[K],lArgs) then
         begin
            if CompatibleArgs(PatternPtr(Items^[k])^.fPrimTypes.Count) then
            begin
               gSuperfluous := dPrimLength-ConstrTypPtr(
                  Constr[coFunctor].Items^[PatternPtr(Items^[K])^.rConstr.Nr]
                                                      )^.nPrimaries.Count;
               gWhichOne := PatternPtr(Items^[K])^.rConstr.Nr;
               with ConstrTypPtr(Constr[coFunctor].Items^[gWhichOne])^ do
                  GetProperties(gProperties);
               with gProperties do
               begin
                  inc(nFirstArg,gSuperfluous);
                  inc(nSecondArg,gSuperfluous);
               end;
            end;
            goto Found;
         end;
      ErrImm(113);
      Found:
         DisposeListOfTerms(lArgs);
   end;
end;
DisposeSubstTrm;
RedefSpecification(ConstrTypPtr(Constr[coFunctor].Items^[gWhichOne])^.fConstrTyp,117{,ikTrmFunctor});
if gWhichOne <> 0 then
begin
   { Robota ponizej jest bledna. Jezeli zaokraglimy typ jako
     typ redefiniowanego funkctora, to:
     - twierdzenie definicyjne moze byc bledne (w twierdzeniu
     definicyjnym wystepuje tylko dolny klaster, wiec wlasciwie
     dowodzimy, ze jezeli cos ma (niektore) wlasnosci wyniku
     funktora (i spelnia definiens) to jest wynkiem funktora,
     a do bazy danych przekazujemy twierdzenie z opuszczeniem
     zalozenia, ze ma te wlasnosci
     - podobnie jest przy dowodzeniu wlasnosci "compatibility":
     dowodzimy rownowaznosc definiensow, przy zalozeniu, ze
     nowy definiens wyznacza funktor, pod warunkiem, ze
     redefiniowany obiekt ma pewne wlasnosci tego funktora
     - wyglada, ze podobnie jest w innych przypadkach

     Chyba tylko przy dowodzeniu "commutativity", mozna skorzystac,
     ze idzie o ten wlasnie funktor !!!!!!!!!!!!

     Dyskusja z Czeskiem, 98.03.12
   }
   { The job below is wrong. If we rounded the type as
     the type of the redefined functor, then:
     - the definition theorem may be incorrect (in the definition
```

```
          theorem there is only the lower cluster, so we are actually
          proving that if something has (some) properties
          of the result of the functor (and satisfies the definiens),
          then it is the result of the functor,
          and we pass the theorem to the database leaving the
          assumption that has these properties
          - it is similar when proving the "compatibility" property:
          we prove the equivalence of definiens, assuming that
          the new definiens is determined by a functor, provided that
          the redefined object has certain properties of this functor
           - it seems to be similar in other cases

          I guess only when proving "commutativity" you can take advantage of the fact
          that this is the functor in question !!!!!!!!!!!!

           Discussion with Czech, 98/03/12}

        (***   lTrm := NewFuncTrm(gWhichOne,C_FormalArgs(dPrimLength));
          lTypPtr := GetTrmType(lTrm);
          lClusterNr := lTypPtr^.UpperCluster;
          dispose(lTypPtr,Done); DisposeTrm(lTrm);
          lCluster.CopyAll(gClusterColl.fItems^[lClusterNr]);
          lCluster.EnlargeBy(gClusterColl.fItems^[ItTyp^.UpperCluster]);
          lCluster.RoundUpWith(ItTyp);
          ItTyp^.UpperCluster := lCluster.CollectCluster;***)
    end;
    if gSpecified then gCorrCond[ord(syCoherence)] := Coherence('K');
    gDefNode.Kind := 'K';
    with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
    begin
       gDefNode.Length := fPrimTypes.Count;
       ReadDefiniens(false,fPrimTypes,ItTyp);
    end;
  end;
```

This code is used in chunk 25.
Defines:
RedefFuncPattern, used in chunk 121.
Uses C_FormalArgs 6d, CheckAccess 31d, Coherence 18a, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

41 ⟨*Parse notation in functor pattern* 41⟩≡

```
  procedure NotatFuncPattern;
  var
     i,K: integer;
     lArgs: TrmList;
     SynonymPattern, OriginPattern: PatternPtr;
     b: integer;
  label Found;
  begin
     gRedef := true;
     InFile.InPos(CurPos);
     InFile.InWord;
     gDefPos := CurPos;
     GetPattern(noFunctor, SynonymPattern);
     CheckAccess(SynonymPattern);
     Notat[noFunctor].InsertExt(SynonymPattern);
     InFile.InWord;

     InFile.InPos(CurPos);
     InFile.InWord;
     GetPattern(noFunctor, OriginPattern);
```

```
    CheckAccess(OriginPattern);
    InFile.InWord;

    fillchar(gSubstTrm,sizeof(gSubstTrm),0);
    gProperties.Properties := [];
    gProperties.nFirstArg := 0;
    gProperties.nSecondArg := 0;
    with OriginPattern^, Notat[noFunctor] do
    begin
       rConstr.Kind := ikTrmFunctor;
       if fFormNr <> 0 then
       begin
          lArgs := CreateConstList(Visible);
          for K := Count-1 downto 0 do
             if (OriginalNr(coFunctor,PatternPtr(Items^[K])^.rConstr.Nr)=0) and
                (PatternPtr(Items^[K])^.fFormNr=fFormNr) and
                CheckTypes(Items^[K],lArgs) then
             begin
                if CompatibleArgs(PatternPtr(Items^[K])^.fPrimTypes.Count) then
                begin
                   gSuperfluous := dPrimLength-ConstrTypPtr(
                      Constr[coFunctor].Items^[PatternPtr(Items^[K])^.rConstr.Nr]
                                                        )^.nPrimaries.Count;
                   gWhichOne := OriginalNr(coFunctor,PatternPtr(Items^[K])^.rConstr.Nr);
                   if gWhichOne = 0 then
                      gWhichOne := PatternPtr(Items^[K])^.rConstr.Nr;
                   with ConstrTypPtr(Constr[coFunctor].Items^[gWhichOne])^ do
                      GetProperties(gProperties);
                   with gProperties do
                   begin
                      inc(nFirstArg,gSuperfluous);
                      inc(nSecondArg,gSuperfluous);
                   end;
                end;
                goto Found;
             end;
          ErrImm(113);
          Found:
             DisposeListOfTerms(lArgs);
       end;
    end;
    DisposeSubstTrm;
    gDefNode.Kind := 'K';

    with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^,rConstr do
    begin
       Kind := ikTrmFunctor;
       Nr := gWhichOne;
       fRedefNr :=  K+1;
    end;

    // with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
    with OriginPattern^ do
    begin
       gDefNode.Length := fPrimTypes.Count;
       {!GB: nie ma definiensu, ale chyba potrzebne sa pewne inicjalizacje!}
       {!GB: no definiens, but I guess some initializations are needed!}
       ReadDefiniens(false,fPrimTypes,ItTyp);
    end;
  end;
```

This code is used in chunk 25.
Defines:
NotatFuncPattern, used in chunk 132.
Uses CheckAccess 31d, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

43a ⟨*Insert functor* 43a⟩≡
```
procedure InsertFunctor;
var
   lConstr: ConstrTypPtr;
   lAbsNr: integer;
begin
   lAbsNr :=  1 + Constr[coFunctor].Count - ConstrBase[coFunctor];
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
   lConstr  :=  new(ConstrTypPtr,
                    InitForPattern(coFunctor,lAbsNr,ArticleID,
                                   fPrimTypes,ItTyp));
   lConstr^.SetProperties(gProperties);
   lConstr^.SetRedef(gWhichOne, gSuperfluous);
   gWhichOne :=  Constr[coFunctor].Count;
   Constr[coFunctor].Insert(lConstr);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_Constructor(lConstr, gWhichOne);
   {$ENDIF}
end;
```
This code is used in chunk 25.
Defines:
InsertFunctor, used in chunk 121.

43b ⟨*Parse definition of functor — tail* 43b⟩≡
```
procedure DefFuncTail;
var
   lPattern: PatternPtr;
begin
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                              rConstr.Nr := gWhichOne;
end;
```
This code is used in chunk 25.
Defines:
DefFuncTail, used in chunk 121.

43c ⟨*Create list of loci* 43c⟩≡
```
procedure CreateLociList(fLowInd,fUpInd: integer; var fTypColl: MCollection);
var
   k: integer;
begin
   fTypColl.Init(fUpInd-fLowInd+1,2);
   for k := fLowInd to fUpInd do
      fTypColl.Insert(gPrimaries[k]^.CopyType);
end;
```
This code is used in chunk 25.
Defines:
CreateLociList, used in chunks 52–54, 58, 59, 62a, and 64.

43d ⟨*Parse mode pattern in definition* 43d⟩≡
```
procedure DefModePattern;
var
   lPattern: PatternPtr;
begin
   InFile.InPos(CurPos);
   InFile.InWord;
   gDefPos := CurPos;
```

```
      GetPattern(noMode, lPattern);
      CheckAccess(lPattern);
      Notat[noMode].InsertExt(lPattern);
      InFile.InWord;
      with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^ do
      begin
         rConstr.Kind := ikTypMode;
         Expansion := nil;
      end;
      Specification;
      { Po co to ??? } {What's this for??? }
      if ItTyp^.TypSort=ikError then ItTyp := AnyTyp^.CopyType;
      gProperties.Properties := [];
      gProperties.nFirstArg := 0;
      gProperties.nSecondArg := 0;
      gDefNode.Kind := 'M';
      with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^ do
      begin
         gDefNode.Length := fPrimTypes.Count;
         ReadDefiniens(false,fPrimTypes,ItTyp);
      end;
      gCorrCond[ord(syExistence)] := Existence('M');
   end;
```

This code is used in chunk 25.
Defines:
DefModePattern, used in chunk 121.
Uses CheckAccess 31d, Existence 20, gDefNode 17a, GetPattern 31b, ReadDefiniens 27c, and Specification 38a.

44 ⟨*Parse mode pattern in redefinition* 44⟩≡

```
   procedure RedefModePattern;
   var
      k,i: integer;
      lArgs: TrmList;
      lPattern: PatternPtr;
   label Found;
   begin
      gRedef := true;
      InFile.InPos(CurPos);
      InFile.InWord;
      gDefPos := CurPos;
      GetPattern(noMode, lPattern);
      CheckAccess(lPattern);
      Notat[noMode].InsertExt(lPattern);
      InFile.InWord;
      fillchar(gSubstTrm,sizeof(gSubstTrm),0);
      gProperties.Properties := [];
      gProperties.nFirstArg := 0;
      gProperties.nSecondArg := 0;
      with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^ do
      begin
         Expansion := nil;
         rConstr.Kind := ikTypMode;
         if fFormNr <> 0 then
         begin
            lArgs := CreateConstList(Visible);
            for k := Count-1 downto 0 do
               if (PatternPtr(Items^[k])^.fFormNr=fFormNr) and
                     (OriginalNr(coMode,PatternPtr(Items^[k])^.rConstr.Nr)=0) and
                     CheckTypes(Items^[k],lArgs) then
               begin
                  if PatternPtr(Items^[k])^.Expansion <> nil then
```

```
                begin
                   ErrImm(134);
                   goto Found;
                end;
                if CompatibleArgs(PatternPtr(Items^[k])^.fPrimTypes.Count) then
                begin
                   gWhichOne := PatternPtr(Items^[k])^.rConstr.Nr;
                   with ConstrTypPtr(Constr[coMode].Items^[gWhichOne])^ do
                      gSuperfluous := dPrimLength-nPrimaries.Count;
                end;
                goto Found;
             end;
          ErrImm(114);
          Found:
             DisposeListOfTerms(lArgs);
       end;
       DisposeSubstTrm;
    end;
    RedefSpecification(ConstrTypPtr(Constr[coMode].Items^[gWhichOne])^.fConstrTyp,118);
    { jaki to ma sens ????????? } {what does that mean????????? }
    if ItTyp^.TypSort=ikError then
    begin
       ItTyp := AnyTyp^.CopyType;
       gWhichOne := 0;
    end;
    if gSpecified then gCorrCond[ord(syCoherence)] := Coherence('M');
    gDefNode.Kind := 'M';
    with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^ do
    begin
       gDefNode.Length := fPrimTypes.Count;
       ReadDefiniens(false,fPrimTypes,ItTyp);
    end;
 end;
```

This code is used in chunk 25.
Defines:
RedefModePattern, used in chunk 121.
Uses CheckAccess 31d, Coherence 18a, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

45 ⟨*Parse mode pattern for notation* 45⟩≡

```
procedure NotatModePattern;
var
   k,i: integer;
   lArgs: TrmList;
   lPattern, origin: PatternPtr;
label Found;
begin
   gRedef := true;
   InFile.InPos(CurPos);
   InFile.InWord;
   gDefPos := CurPos;
   GetPattern(noMode, lPattern);
   CheckAccess(lPattern);
   InFile.InWord;
   InFile.InPos(CurPos);
   InFile.InWord;
   GetPattern(noMode, origin);
   CheckAccess(origin);
   fillchar(gSubstTrm,sizeof(gSubstTrm),0);
   with Notat[noMode], origin^ do
   begin
```

```
        Expansion := nil;
        rConstr.Kind := ikTypMode;
        if fFormNr<>0 then
        begin
           lArgs := CreateConstList(Visible);
           for k := Count-1 downto 0 do
              if (PatternPtr(Items^[k])^.fFormNr=fFormNr) and
                    (OriginalNr(coMode,PatternPtr(Items^[k])^.rConstr.Nr)=0) and
                    CheckTypes(Items^[k],lArgs) then
              begin
                 if PatternPtr(Items^[k])^.Expansion <> nil then
                 begin
                    ErrImm(134);
                    goto Found;
                 end;
                 if CompatibleArgs(PatternPtr(Items^[k])^.fPrimTypes.Count) then
                 begin
                    gWhichOne := OriginalNr(coMode,PatternPtr(Items^[k])^.rConstr.Nr);
                    if gWhichOne = 0 then
                       gWhichOne := PatternPtr(Items^[k])^.rConstr.Nr;
                    with ConstrTypPtr(Constr[coMode].Items^[gWhichOne])^ do
                       gSuperfluous := dPrimLength- nPrimaries.Count;
                 end;
                 goto Found;
              end;
           ErrImm(114);
           Found:
              DisposeListOfTerms(lArgs);
        end;
        DisposeSubstTrm;
     end;
     {GB: Nie ma specyfikacji, ale moze sie przydac jakas inicjalizacja}
     {GB: No specification, but some initialization might be useful}
     RedefSpecification(ConstrTypPtr(Constr[coMode].Items^[gWhichOne])^.fConstrTyp,118);
     { jaki to ma sens ????????? } {what does that mean????????? }
     if ItTyp^.TypSort=ikError then
     begin
        ItTyp := AnyTyp^.CopyType;
        gWhichOne := 0;
     end;
     gDefNode.Kind := 'M';
     with origin^ do
     begin
        gDefNode.Length := fPrimTypes.Count;
        ReadDefiniens(false,fPrimTypes,ItTyp);
     end;
     Notat[noMode].InsertExt(lPattern);
     InFile.InWord;
     with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^, rConstr do
     begin
        Expansion := nil;
        Kind := ikTypMode;
        Nr := gWhichOne;
        fRedefNr :=  K+1;
     end;
  end;
```

This code is used in chunk 25.
Defines:
NotatModePattern, used in chunk 132.
Uses CheckAccess 31d, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

47a ⟨*Insert a mode* 47a⟩≡

```
procedure InsertMode;
var
   lConstr: ConstrTypPtr;
   lAbsNr: integer;
begin
   lAbsNr :=  1 + Constr[coMode].Count - ConstrBase[coMode];
   with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^ do
   lConstr := new(ConstrTypPtr,
                  InitForPattern(coMode,lAbsNr,ArticleID,fPrimTypes,ItTyp));
   lConstr^.SetProperties(gProperties);

   if Constr[coMode].Count>1 then {Checking if we are not processing HIDDEN accommodated with the -h flag}
      if (ItTyp^.TypSort = ikTypMode) and
            (sySethood in ConstrTypPtr(Constr[coMode].Items^[ItTyp^.ModNr])^.fProperties) then
         lConstr^.fProperties := lConstr^.fProperties+[sySethood]
      else if (gWhichOne <> 0) and
                (sySethood in ConstrTypPtr(Constr[coMode].Items^[gWhichOne])^.fProperties) then
         lConstr^.fProperties := lConstr^.fProperties+[sySethood];
   lConstr^.SetRedef(gWhichOne, gSuperfluous);
   gWhichOne :=  Constr[ coMode].Count;
   Constr[ coMode].Insert(lConstr);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_Constructor(lConstr, gWhichOne);
   {$ENDIF}
end;
```

This code is used in chunk 25.
Defines:
`InsertMode`, used in chunk 121.

47b ⟨*Parse definition of an expandable mode* 47b⟩≡

```
procedure DefExpandableMode;
var
   lPattern: PatternPtr;
begin
   InFile.InPos(CurPos);
   InFile.InWord;
   gDefPos := CurPos;
   GetPattern(noMode, lPattern);
   CheckAccess(lPattern);
   Notat[noMode].InsertExt(lPattern);
   with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^, rConstr do
   begin
      Kind := ikTypMode;
      Nr := 0;
      Expansion := ReadType;
      if Expansion^.TypSort=ikError then Expansion := AnyTyp^.CopyType;
      Infile.InWord;
      Expansion^.WithinType(ChangeToLoci);
   end;
end;
```

This code is used in chunk 25.
Defines:
`DefExpandableMode`, used in chunk 121.
Uses `CheckAccess` 31d, `GetPattern` 31b, and `ReadType` 7b.

47c ⟨*Parse predicate or attribute pattern in definition* 47c⟩≡

```
procedure DefPredAttributePattern;
var
   lPattern: PatternPtr;
begin
```

```
    InFile.InPos(CurPos);
    InFile.InWord;
    gDefPos := CurPos;
    GetPattern(noAttribute, lPattern);
    CheckAccess(lPattern);
    Notat[noAttribute].InsertExt(lPattern);
    InFile.InWord;
    RedefAntonym := false;
    with Notat[noAttribute], PatternPtr(Items^[Count+fExtCount-1])^ do
    begin
       rConstr.Kind := ikFrmAttr;
       gDefNode.Kind := 'V';
       gDefNode.Length := fPrimTypes.Count;
       ReadDefiniens(false,fPrimTypes,nil);
    end;
  end;
```

This code is used in chunk 25.
Defines:
DefPredAttributePattern, used in chunk 121.
Uses CheckAccess 31d, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

48 ⟨*Parse pattern for predicate or attribute redefinition* 48⟩≡

```
  procedure RedefPredAttributePattern;
  var
     k,i: integer;
     lArgs: TrmList;
     lPattern: PatternPtr;
  label Found;
  begin
     gRedef := true;
     InFile.InPos(CurPos); InFile.InWord;
     gDefPos := CurPos;
     GetPattern(noAttribute, lPattern);
     CheckAccess(lPattern);
     Notat[noAttribute].InsertExt(lPattern);
     fillchar(gSubstTrm,sizeof(gSubstTrm),0);
     InFile.InWord;
     RedefAntonym := false;
     with Notat[noAttribute], PatternPtr(Items^[Count+fExtCount-1])^ do
     begin
        rConstr.Kind := ikFrmAttr;
        gDefNode.Positive := true;
        if fFormNr<>0 then
        begin
           lArgs := CreateConstList(Visible);
           for K := Count-1 downto 0 do
              if (OriginalNr(coAttribute,PatternPtr(Items^[K])^.rConstr.Nr)=0) and
                    (PatternPtr(Items^[K])^.fFormNr=fFormNr) and
                    CheckTypes(Items^[k],lArgs) then
              begin
                 if PatternPtr(Items^[K])^.fAntonymic then RedefAntonym := true;
                 if CompatibleArgs(PatternPtr(Items^[K])^.fPrimTypes.Count) then
                 begin
                    gSuperfluous := dPrimLength-ConstrTypPtr(
                       Constr[ coAttribute].Items^[PatternPtr(Items^[K])^.rConstr.Nr]
                                                          )^.nPrimaries.Count;
                    gWhichOne := PatternPtr(Items^[K])^.rConstr.Nr;
                    gDefNode.Positive := not PatternPtr(Items^[K])^.fAntonymic;
                 end;
                 goto Found
              end;
```

```
            ErrImm(115);
            Found:
               DisposeListOfTerms(lArgs);
         end;
      end;
      DisposeSubstTrm;
      gDefNode.Kind := 'V';
      with Notat[noAttribute], PatternPtr(Items^[Count+fExtCount-1])^ do
      begin
         gDefNode.Length := fPrimTypes.Count;
         ReadDefiniens(RedefAntonym,fPrimTypes,nil);
      end;
   end;
```

This code is used in chunk 25.
Defines:
RedefPredAttributePattern, used in chunk 121.
Uses CheckAccess 31d, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

49 ⟨*Parse notation in a predicate or attribute pattern* 49⟩≡

```
   procedure NotatPredAttributePattern;
   var
      k,i: integer;
      lArgs: TrmList;
      SynonymPattern, OriginPattern: PatternPtr;
      Antonymic, lSynonym: boolean;
   label Found;
   begin
      gRedef := true;
      InFile.InPos(CurPos);
      InFile.InWord;
      gDefPos := CurPos;
      GetPattern(noAttribute, SynonymPattern);
      CheckAccess(SynonymPattern);
      Notat[noAttribute].InsertExt(SynonymPattern);

      InFile.InWord;
      lSynonym := InFile.Current.Kind = ikMscSynonym;
      InFile.InWord;
      InFile.InPos(CurPos); InFile.InWord;
      GetPattern(noAttribute, OriginPattern);
      CheckAccess(OriginPattern);

      fillchar(gSubstTrm,sizeof(gSubstTrm),0);
      InFile.InWord;
      RedefAntonym := false;
      with Notat[noAttribute], OriginPattern^ do
      begin
         rConstr.Kind := ikFrmAttr;
         gDefNode.Positive := true;
         if fFormNr<>0 then
         begin
            lArgs := CreateConstList(Visible);
            for K := Count-1 downto 0 do
               if (OriginalNr(coAttribute,PatternPtr(Items^[K])^.rConstr.Nr)=0) and
                     (PatternPtr(Items^[K])^.fFormNr=fFormNr) and
                     CheckTypes(Items^[k],lArgs) then
               begin
                  if PatternPtr(Items^[K])^.fAntonymic then RedefAntonym := true;
                  if CompatibleArgs(PatternPtr(Items^[K])^.fPrimTypes.Count) then
                  begin
                     gSuperfluous := dPrimLength-ConstrTypPtr(
```

```
                  Constr[ coAttribute].Items^[PatternPtr(Items^[K])^.rConstr.Nr]
                                                        )^.nPrimaries.Count;
               gWhichOne := OriginalNr(coAttribute,PatternPtr(Items^[K])^.rConstr.Nr);
               if gWhichOne = 0 then
                  gWhichOne := PatternPtr(Items^[K])^.rConstr.Nr;
               gDefNode.Positive :=  not PatternPtr(Items^[K])^.fAntonymic;
            end;
            goto Found
         end;
      ErrImm(115);
      Found:
         DisposeListOfTerms(lArgs);
   end;
  end;
  DisposeSubstTrm;
  gDefNode.Kind := 'V';
  with SynonymPattern^, rConstr do
  begin
     Kind := ikFrmAttr;
     Nr := gWhichOne;
     fRedefNr :=  K+1;
     fAntonymic := lSynonym = RedefAntonym;
  end;
  with OriginPattern^ do
  begin
     gDefNode.Length := fPrimTypes.Count;
     ReadDefiniens(RedefAntonym,fPrimTypes,nil);
  end;
end;
```

This code is used in chunk 25.
Defines:
NotatPredAttributePattern, used in chunk 132.
Uses CheckAccess 31d, CompatibleArgs 16b, CreateConstList 34a, gDefNode 17a, GetPattern 31b, and ReadDefiniens 27c.

50 ⟨*Insert predicate or attribute* 50⟩≡

```
// ##NOTE: the Abstract property is by default false
procedure InsertPredAttribute;
var
   lTypPtr: TypPtr;
   lConstr: ConstrTypPtr;
   lAbsNr: integer;
begin
   lAbsNr :=  1 + Constr[coAttribute].Count - ConstrBase[coAttribute];
   if gPrimNbr > 0 then lTypPtr := gPrimaries[gPrimNbr]^.CopyType
   else lTypPtr := NewIncorTyp;
   with Notat[noAttribute], PatternPtr(Items^[Count+fExtCount-1])^ do
   lConstr :=  new(ConstrTypPtr,
                   InitForPattern(coAttribute,lAbsNr,ArticleID,
                                  fPrimTypes,lTypPtr));
   lConstr^.SetRedef(gWhichOne, gSuperfluous);
   gWhichOne :=  Constr[coAttribute].Count;
   Constr[coAttribute].Insert(lConstr);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_Constructor(lConstr, gWhichOne);
   {$ENDIF}
end;
```

This code is used in chunk 25.
Defines:
InsertPredAttribute, used in chunk 121.

51a ⟨*Analyze cluster* 51a⟩≡

```
procedure AnalyzeCluster(const fList: MCollection;
                         var fAttrs: MCollection;
                         fTyp:TypPtr);
var
   lAttr: AttrPtr;
   lTyp: TypPtr;
   z: integer;
begin
   lTyp := fTyp^.CopyType;
   fAttrs.Init(0,10);
   with fList do
      for z := Count-1 downto 0 do
      begin
         lAttr := AnalyzeAttribute(AttrNodePtr(Items^[z]),lTyp);
         if lAttr = nil then
         begin
            if (lTyp^.TypSort<>ikTypError) and (AttrNodePtr(Items^[z])^.nInt<>0) then
               Error(AttrNodePtr(Items^[z])^.nPos,115);
            exit;
         end;
         if AttrNodePtr(Items^[z])^.nNeg then
            if lAttr^.fNeg = 0 then
               lAttr^.fNeg := 0
            else lAttr^.fNeg := 1
            else if lAttr^.fNeg = 0 then
               lAttr^.fNeg := 1
            else lAttr^.fNeg := 0;
         //  lTyp^.LowerCluster^.Insert(lAttr^.CopyAttribute);
         if not lTyp^.LowerCluster^.fConsistent then
         begin
            Error(AttrNodePtr(Items^[z])^.nPos,95);
            lTyp^.TypSort := ikError;
            dispose(lTyp,Done);
            dispose(lAttr,Done); //!!
            exit
         end;
         //  lTyp^.UpperCluster^.Insert(lAttr^.CopyAttribute);
         //  lTyp^.RoundUp;
         //  fAttrs.Insert(lAttr);
         fAttrs.AtInsert(0,lAttr);
      end;
   dispose(lTyp,Done);
end;
```

This code is used in chunk 25.
Defines:
`AnalyzeCluster`, used in chunks 52–54.

51b ⟨*Add items to a cluster* 51b⟩≡

```
procedure AddToCluster(var fList: MList; fCluster: AttrCollectionPtr);
var
   lAttr: AttrPtr;
   z: integer;
begin
   with fList do
      for z := 0 to Count-1 do
         fCluster^.Insert(Items^[z]);
   if not fCluster^.fConsistent then ErrImm(95);
   fList.DeleteAll;
   fList.Done;
```

```
end;
```

This code is used in chunk 25.
Defines:
AddToCluster, used in chunks 52–54.

52 ⟨*Define existential cluster* 52⟩≡

```
procedure DefExistentialCluster;
var
   lTyp,llTyp: TypPtr;
   lList: MCollection;
   lClusterPtr: AttrCollectionPtr;
   lAttrFrm: AttributiveFormula;
   lFrm: FrmPtr;
   lTypList,lAttrs: MCollection;
   lAbsNr: integer;
   lErrorOcc: boolean;
begin
   lErrorOcc := false;
   InFile.InPos(CurPos);
   LoadIPNColl(lList);
   lTyp := ReadType;
   Infile.InWord;
   llTyp := AdjustedType(lTyp);
   BoundVarNbr := 1;
   BoundVar[1] := lTyp;
   lAttrFrm.Init(ikFrmAttr,new(SimpleTermPtr, Init('B',1)),CurPos,lList);
   lFrm := lAttrFrm.Analyze;
   lClusterPtr := CopyCluster(llTyp^.LowerCluster);
   AnalyzeCluster(lList,lAttrs,lTyp);
   AddToCluster(lAttrs,lClusterPtr);
   gCorrCond[ord(syExistence)] := NewNegDis(NewUniv(lTyp,NewNegDis(lFrm)));
   lAttrFrm.Done;
   CreateLociList(1,dPrimLength,lTypList);
   InitAccess;
   InitLociForCluster(lClusterPtr);
   if not AllLociAccessibleInTyp(lTypList,llTyp) then
   begin
      dispose(lClusterPtr,Done);
      lTypList.Done;
      BoundVarNbr := 0;
      lErrorOcc := true;
      ErrImm(100);
   end;
   if (llTyp^.TypSort = ikError) or gConstInExportableItemOcc then lErrorOcc := true;
   if not lErrorOcc then
   begin
      lAbsNr :=  1 + RegisteredCluster.Count +  RegisteredCluster.fExtCount - RegClusterBase;
      RegisteredCluster.InsertExt(new(RClusterPtr,
                                      RegisterCluster(lAbsNr,ArticleID,lClusterPtr,lTypList,llTyp)));
      {$IFDEF ANALYZER_REPORT}
      with RegisteredCluster do
         AReport.Out_RCluster(Items^[Count+fExtCount-1]);
      {$ENDIF}
   end
   else
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_ErrCluster(elRCluster);
      {$ENDIF}
   end;
```

```
    dispose(llTyp,Done);
    { implementacje nalezy poprawic, typ jest dwukrotnie
      adjustowany
    }
    { implementations need to be improved, type is twice
       adjusted
     }
    BoundVarNbr := 0;
  end;
```

This code is used in chunk 25.
Defines:
DefExistentialCluster, used in chunk 129.
Uses AddToCluster 51b, AllLociAccessibleInTyp 32b, Analyze 138, AnalyzeCluster 51a, CreateLociList 43c, InitAccess 31c, InitLociForCluster 32a, and ReadType 7b.

53 ⟨*Define conditional cluster* 53⟩≡

```
  procedure DefConditionalCluster;
  var
    lTyp,llTyp: TypPtr;
    lList,lList1: MCollection;
    lClusterPtr,lClusterPtr1: AttrCollectionPtr;
    lAttrFrm1,lAttrFrm: AttributiveFormula;
    lFrm,lFrm1: FrmPtr;
    lTypList,lAttrs1,lAttrs2: MCollection;
    lAbsNr: integer;
    lErrorOcc: boolean;
  begin
    lErrorOcc := false;
    InFile.InPos(CurPos);
    LoadIPNColl(lList);
    LoadIPNColl(lList1);
    lTyp := ReadType;
    llTyp := AdjustedType(lTyp);
    lClusterPtr := CopyCluster(llTyp^.LowerCluster);
    AnalyzeCluster(lList,lAttrs1,lTyp);
    AddToCluster(lAttrs1,lClusterPtr);
    lClusterPtr1 := CopyCluster(llTyp^.LowerCluster);
    AnalyzeCluster(lList1,lAttrs2,lTyp);
    AddToCluster(lAttrs2,lClusterPtr1);
    BoundVarNbr := 1;
    BoundVar[1] := lTyp;
    lAttrFrm.Init(ikFrmAttr,new(SimpleTermPtr, Init('B',1)),CurPos,lList);
    lFrm := lAttrFrm.Analyze;
    lAttrFrm.Done;
    lAttrFrm1.Init(ikFrmAttr,new(SimpleTermPtr, Init('B',1)),CurPos,lList1);
    lFrm1 := lAttrFrm1.Analyze;
    lAttrFrm1.Done;
    gCorrCond[ord(syCoherence)] := NewUniv(lTyp,NewImpl(lFrm,lFrm1));
    dispose(llTyp^.LowerCluster,Done);
    llTyp^.LowerCluster := NewEmptyCluster;
    dispose(llTyp^.UpperCluster,Done);
    llTyp^.UpperCluster := NewEmptyCluster;
    CreateLociList(1,dPrimLength,lTypList);
    InitAccess;
    InitLociForCluster(lClusterPtr);
    if not AllLociAccessibleInTyp(lTypList,llTyp) then
    begin
      dispose(lClusterPtr,Done);
      dispose(lClusterPtr1,Done);
      lTypList.Done;
      lErrorOcc := true;
```

```
      ErrImm(100);
    end;
    if (llTyp^.TypSort = ikError) or gConstInExportableItemOcc then lErrorOcc := true;
    if not lErrorOcc then
    begin
      lAbsNr :=  1 + ConditionalCluster.Count + ConditionalCluster.fExtCount - CondClusterBase;
      ConditionalCluster.InsertExt(new(CClusterPtr,
                                       RegisterCluster(lAbsNr,ArticleID,lClusterPtr,lClusterPtr1,lTypList,llTyp))
      {$IFDEF ANALYZER_REPORT}
      with ConditionalCluster do
        AReport.Out_CCluster(ConditionalCluster.Items^[Count+fExtCount-1]);
      {$ENDIF}
    end
    else
    begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_ErrCluster(elCCluster);
      {$ENDIF}
    end;
    Infile.InWord;
    dispose(llTyp,Done);
    BoundVarNbr := 0;
  end;
```

This code is used in chunk 25.
Defines:
DefConditionalCluster, used in chunk 129.
Uses AddToCluster 51b, AllLociAccessibleInTyp 32b, Analyze 138, AnalyzeCluster 51a, CreateLociList 43c, InitAccess 31c, InitLociForCluster 32a, and ReadType 7b.

54 ⟨*Define functorial cluster* 54⟩≡

```
  procedure DefFunctorCluster;
  var
    lTerm: ExpPtr;
    lAttrFrm: AttributiveFormula;
    lFrm,lFrm1: FrmPtr;
    lClusterPtr: AttrCollectionPtr;
    lTrm,llTrm:TrmPtr;
    lTyp,llTyp: TypPtr;
    lList,lAttrs,lConjuncts,lTypList: MCollection;
    A: TrmList;
    lFuncNr,lAbsNr,zz,z,i: integer;
    lErrorOcc: boolean;
  begin
    lErrorOcc := false;
    InFile.InPos(CurPos); BoundVarNbr := 0;
    lTerm := LoadTerm;
    lTrm := lTerm^.Analyze;
    dispose(lTerm,Done);
    if not (lTrm^.TrmSort in [ikTrmFunctor,ikTrmSelector,ikTrmAggreg,ikTrmError]) then
    begin
      ErrImm(96);
      lErrorOcc := true;
      dispose(lTrm,Done);
      lTrm := NewInCorTrm;
    end;
    if lTrm^.TrmSort = ikTrmFunctor then
    begin
      AdjustTrm(lTrm,lFuncNr,A);
      llTrm := NewFuncTrm(lFuncNr,CopyTermList(A));
      //   llTrm^.nPattNr := lTrm^.nPattNr;
    end
```

```
    else llTrm := CopyTerm(lTrm);
    LoadIPNColl(lList);
    InFile.InWord;
    llTyp := nil;
    if InFile.Current.Kind = '.' then
    begin
       llTyp := ReadType;
       if llTyp^.TypSort = ikError then
       begin
          llTyp := nil;
          lTyp := GetTrmType(llTrm);
       end
       else lTyp := llTyp^.CopyType;
       InFile.InWord;
    end
    else lTyp := GetTrmType(llTrm);
    lClusterPtr := CopyCluster(lTyp^.LowerCluster);
    AnalyzeCluster(lList,lAttrs,lTyp);
    if llTyp <> nil then
    begin
 {-----\/----- EXCLUDED -----\/----- EnlargeBy does not seem to work properly!!!
    lTyp.LowerCluster.EnlargeBy(@lAttrs);
    lTyp.UpperCluster.EnlargeBy(lTyp.LowerCluster);
  -----/\----- EXCLUDED -----/\-----}
 {    for zz := 0 to lAttrs.Count-1 do
        begin
           lTyp.LowerCluster.Insert(AttrPtr(lAttrs.Items^[zz])^.CopyAttribute);
           lTyp.UpperCluster.Insert(AttrPtr(lAttrs.Items^[zz])^.CopyAttribute);
        end;
    gCorrCond[ord(syCoherence)] := NewQualFrm(lTrm,lTyp);}
       BoundVarNbr := 1;
       BoundVar[1] := lTyp;
       lFrm := NewEqFrm(NewVarTrm(ikTrmBound,1),CopyTerm(lTrm));
       lConjuncts.Init(lAttrs.Count,2);
       with lAttrs do
          for z := 0 to Count-1 do
             with AttrPtr(Items^[z])^ do
          begin
             lFrm1 := NewPredFrm(ikFrmAttr,fAttrNr,
                                 AddToTrmList(CopyTermList(fAttrArgs),
                                              NewVarTrm(ikTrmBound,1)),0);
             if fNeg = 0 then
                lFrm1 := NewNeg(lFrm1);
             lConjuncts.Insert(lFrm1);
          end;
       gCorrCond[ord(syCoherence)] := NewUniv(lTyp^.CopyType,
                                              NewImpl(lFrm,NewConjFrm(lConjuncts)));
       BoundVarNbr := 0;
    end
    else
    begin
       dispose(lTyp,Done);
       lConjuncts.Init(lAttrs.Count,2);
       with lAttrs do
          for z := 0 to Count-1 do
             with AttrPtr(Items^[z])^ do
          begin
             lFrm := NewPredFrm(ikFrmAttr,fAttrNr,
                                AddToTrmList(CopyTermList(fAttrArgs),CopyTerm(lTrm)),0);
             if fNeg = 0 then
```

```
                lFrm := NewNeg(lFrm);
              lConjuncts.Insert(lFrm);
           end;
        DisposeTrm(lTrm);
        gCorrCond[ord(syCoherence)] := NewConjFrm(lConjuncts);
     end;
     AddToCluster(lAttrs,lClusterPtr);
     CreateLociList(1,dPrimLength,lTypList);
     InitAccess;
     if not AllLociAccessibleInTrm(lTypList,llTrm) then
     begin
        dispose(lClusterPtr,Done);
        lTypList.Done;
        lErrorOcc := true;
        ErrImm(100);
     end;
     if (llTrm^.TrmSort = ikError) or gConstInExportableItemOcc then lErrorOcc := true;
     if not lErrorOcc then
     begin
        lAbsNr :=  1 + FunctorCluster.Count +  FunctorCluster.fExtCount - FuncClusterBase;
        // ##TODO: since Preparator makes use of the cluster immediately,
        //         this should rather be normal Insert. Any problem with that?
        FunctorCluster.InsertExt(new(FClusterPtr,
                                     RegisterCluster(lAbsNr,ArticleID,lClusterPtr,lTypList,llTrm,llTyp)));
        {$IFDEF ANALYZER_REPORT}
        with FunctorCluster do
           AReport.Out_FCluster(FunctorCluster.Items^[Count+fExtCount-1]);
        {$ENDIF}
     end
     else
     begin
        {$IFDEF ANALYZER_REPORT}
        AReport.Out_ErrCluster(elFCluster);
        {$ENDIF}
     end;
     DisposeTrm(llTrm);
     if llTyp<> nil
     then dispose(llTyp,Done);
  end;
```

This code is used in chunk 25.
Defines:
  DefFunctorCluster, used in chunk 129.
Uses AddToCluster 51b, AllLociAccessibleInTrm 33a, Analyze 138, AnalyzeCluster 51a, CreateLociList 43c, InitAccess 31c, and ReadType 7b.

56a ⟨*Collect loci* 56a⟩≡

```
  var gLociSet: NatSet;
  procedure CollectLoci(var fTrm: TrmPtr);
  begin
     with VarTrmPtr(fTrm)^ do
        if TrmSort = ikTrmLocus then gLociSet.InsertElem(VarNr);
  end;
```

This code is used in chunk 25.
Defines:
  CollectLoci, used in chunk 56b.
  gLociSet, used in chunk 56b.

56b ⟨*Find pattern* 56b⟩≡

```
  procedure FindPattern(aKind: char; var aIdData: _IdentifyData);
  var
     i,k,lFormNr,lConstrNr,lPattNr: integer;
```

```
   lAntonymic: boolean;
   lArgs, lTrmList: TrmList;
   lTyp: TypPtr;
   lVisible: IntSequence;
label Found;
begin
   with aIdData do
   begin
      Err := false;
      ReadPattern(lFormNr,lVisible);
      VisibleCnt := lVisible.fCount;
      if lFormNr = 0 then Err := true;
      for k := 0 to lVisible.fCount-1 do
         if lVisible.fList^[k] = 0 then Err := true;
      InFile.InWord;
      with Notat[NotatKind(aKind)]  do
      begin
         lConstrNr := 0;
         fillchar(gSubstTrm,sizeof(gSubstTrm),0);
         lAntonymic := false;
         if lFormNr <> 0 then
         begin
            lArgs := CreateTrmList(lVisible);
            if lArgs <> InCorrTrmList then
            begin
               for k := Count-1 downto 0 do
                  if (PatternPtr(Items^[k])^.fFormNr = lFormNr) and
                        CheckTypes(Items^[k],lArgs) then
                  begin
                     lConstrNr := PatternPtr(Items^[k])^.rConstr.Nr;
                     lAntonymic := PatternPtr(Items^[k])^.fAntonymic;
                     lPattNr := k;
                     goto Found;
                  end;
               ErrImm(113);
            end;
            Err := true;
            Found:
               DisposeListOfTerms(lArgs);
         end;
         /// the set all loci accessible from visible arguments
         gLociSet.Init(0,MaxArgNbr);
         for k := 0 to lVisible.fCount-1 do
            if lVisible.fList^[k] <> 0 then
            begin
               gLociSet.InsertElem(lVisible.fList^[k]);
               gPrimaries[lVisible.fList^[k]-gDefBase]^.WithinType(CollectLoci);
            end;
         ArgsSet.MoveNatSet(gLociSet);
         lVisible.Done;
         // the Pattern
         lTrmList := CreateArgList1;
         case aKind of
            ikTrmFunctor: Pattern := NewFuncTrm(lConstrNr,lTrmList);
            ikFrmPred:
               begin
                  Pattern := NewPredFrm(ikFrmPred,lConstrNr,lTrmList,1+lPattNr);
                  if lAntonymic then Pattern := NewNeg(FrmPtr(Pattern));
               end;
            ikFrmAttr:
```

```
                begin
                   Pattern := NewPredFrm(ikFrmAttr,lConstrNr,lTrmList,1+lPattNr);
                   if lAntonymic then Pattern := NewNeg(FrmPtr(Pattern));
                end;
             end;
          end;
       end;
    end;
```

This code is used in chunk 25.
Defines:
_IdentifyData, used in chunk 59.
FindPattern, used in chunk 59.
Uses CollectLoci 56a, CreateTrmList 34b, gLociSet 56a, and ReadPattern 30b.

58 ⟨*Define reduction* 58⟩≡

```
procedure DefReduction;
var
   lTerm1,lTerm2: ExpPtr;
   lTrm1,lTrm2: TrmPtr;
   lTypList: MCollection;
   lAbsNr: integer;
   lErrorOcc: boolean;
   lPos,lTermPos: Position;
   lReduction: ReductionPtr;
begin
   lErrorOcc := false;
   InFile.InPos(CurPos);
   BoundVarNbr := 0; // po co?
   lTerm1 := LoadTerm;
   lTerm2 := LoadTerm;
   lTrm1 := lTerm1^.Analyze;
   lTrm2 := lTerm2^.Analyze;
   lTermPos := CurPos;
   {$IFDEF MDEBUG}
   InfoString('Reduction Start'); InfoNewLine;
   InfoTerm(lTrm1); InfoNewLine;
   InfoTerm(lTrm2); InfoNewLine;
   InfoString('Reduction End'); InfoNewLine;
   {$ENDIF}
   dispose(lTerm1,Done);
   dispose(lTerm2,Done);
   if not ReductionAllowed(lTrm1,lTrm2) then
   begin
      ErrImm(257);
      lErrorOcc := true;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elReduction);
      AReport.Out_XAttr(atAid, ArticleID);
      AReport.Out_XIntAttr(atNr, 0);
      AReport.Out_XAttrEnd;
      AReport.Out_XEl1(elErrorReduction);
      AReport.Out_XElEnd(elReduction);
      {$ENDIF}
   end;
   InFile.InWord; InFile.InPos(lPos);
   gCorrCond[ord(syReducibility)] := NewIncorFrm;
   if lErrorOcc then exit;
   CreateLociList(1,dPrimLength,lTypList);
   InitAccess;
   if not AllLociAccessibleInTrm(lTypList,lTrm1) then Error(lTermPos,100);
   gCorrCond[ord(syReducibility)] := NewEqFrm(CopyTerm(lTrm1),CopyTerm(lTrm2));
```

```
    WithinTerm(lTrm1,ChangeToLoci);
    WithinTerm(lTrm2,ChangeToLoci);
    lReduction := new(ReductionPtr,
                              Init(1+gReductions.Count, ArticleID,
                                   lTypList,lTrm1,lTrm2));
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_Reduction(lReduction);
    {$ENDIF}
    gReductions.Insert(lReduction);
  end;
```

This code is used in chunk 25.
Defines:
DefReduction, used in chunk 129.
Uses AllLociAccessibleInTrm 33a, Analyze 138, CreateLociList 43c, and InitAccess 31c.

59 ⟨*Define identify* 59⟩≡

```
  procedure DefIdentify(aKind: char);
  var
    k,lNr,rNr: integer;
    lId, rId: _IdentifyData;
    lEqArgs: IntRel;
    lConjuncts: MCollection;
    lFrm: FrmPtr;
    lAllArgsSet,ldSet,rdSet,lCommonArgs,lArgs: NatSet;
    lIdentify: IdentifyPtr;
    lErrIdentify: boolean;
    lTypList: MCollection;
    lIdPattern,rIdPattern:ExprPtr;
  begin
    lErrIdentify := false;
    InFile.InPos(CurPos); InFile.InWord;
    FindPattern(aKind,lId);
    if lId.Err then lErrIdentify := true;
    FindPattern(aKind,rId);
    if rId.Err then lErrIdentify := true;
    lAllArgsSet.CopyNatSet(lId.ArgsSet);
    lAllArgsSet.EnlargeBy(rId.ArgsSet);
    for k := dPrimLength-lAllArgsSet.Count downto 1 do
    begin ErrImm(100); lErrIdentify := true end;
    lAllArgsSet.Done;
    lCommonArgs.CopyNatSet(lId.ArgsSet);
    lCommonArgs.IntersectWith(rId.ArgsSet);
    //  Left pattern and right pattern arguments
    ldSet.CopyNatSet(lId.ArgsSet); ldSet.ComplementOf(rId.ArgsSet);
    rdSet.CopyNatSet(rId.ArgsSet); rdSet.ComplementOf(lId.ArgsSet);
    // "when"
    lEqArgs.Init(0);
    while InFile.Current.Kind <> ';' do
    begin
      lNr := InFile.Current.Nr; InFile.InPos(CurPos); InFile.InWord;
      rNr := InFile.Current.Nr; InFile.InPos(CurPos); InFile.InWord;
      if (lNr = 0) or (rNr = 0) then
        lErrIdentify := true;
      if ldSet.ElemNr(lNr) >= 0 then
      begin
        lEqArgs.AssignPair(lNr,rNr);
        if ldSet.ElemNr(rNr) >= 0 then
        begin ErrImm(98); lErrIdentify := true end
        else if not rdSet.ElemNr(rNr) >=0 then
        begin ErrImm(99); lErrIdentify := true end;
        // checking arguments type
```

```
      if not lErrIdentify and
            not FixedVar[rNr].nTyp^.IsWiderThan(FixedVar[lNr].nTyp^.CopyType) then
      begin ErrImm(139); lErrIdentify := true end;
   end
   else if rdSet.ElemNr(lNr) >= 0 then
   begin
      lEqArgs.AssignPair(rNr,lNr);
      if rdSet.ElemNr(rNr) >= 0 then
      begin ErrImm(98); lErrIdentify := true end
      else if not ldSet.ElemNr(rNr) >=0 then
      begin ErrImm(99); lErrIdentify := true end;
      // checking arguments type
      if not lErrIdentify and
            not FixedVar[lNr].nTyp^.IsWiderThan(FixedVar[rNr].nTyp^.CopyType) then
      begin ErrImm(139); lErrIdentify := true end;
   end
   else
   begin ErrImm(99); lErrIdentify := true end;
end;
InFile.InWord;
// checking (visible) arguments correcnesss
for k  :=  0 to lEqArgs.Count - 1 do
   lCommonArgs.InsertElem(lEqArgs.Items^[k].X);
InitAccess;
for k  :=  0 to lCommonArgs.Count - 1 do
   LociOcc[lCommonArgs.Items^[k].X] := true;
for k := 1 to dPrimLength do
   if LociOcc[k] then
      gPrimaries[k]^.WithinType(SetLociOcc);
lArgs.CopyNatSet(lCommonArgs);
for k := 1 to dPrimLength do
   if LociOcc[k] then
      lArgs.InsertElem(k);
if not lArgs.IsEqualTo(lId.ArgsSet) then
begin ErrImm(189); lErrIdentify := true end;
// Correctness condition: compatibility
if lErrIdentify then
   gCorrCond[ord(syCompatibility)] := NewIncorFrm
else if lEqArgs.Count = 0 then
   case aKind of
      ikTrmFunctor:
         gCorrCond[ord(syCompatibility)] := NewEqFrm(TrmPtr(lId.Pattern),TrmPtr(rId.Pattern));
      ikFrmPred,ikFrmAttr:
         gCorrCond[ord(syCompatibility)] := NewBicond(FrmPtr(lId.Pattern),FrmPtr(rId.Pattern));
   end
else
begin
   lConjuncts.Init(lEqArgs.Count,2);
   for k := 0 to lEqArgs .Count-1 do
      with lEqArgs .Items^[k] do
         lConjuncts.Insert(NewEqFrm(NewVarTrm(ikTrmConstant,X),NewVarTrm(ikTrmConstant,Y)));
   if lConjuncts.Count = 1 then
   begin
      lFrm := FrmPtr(lConjuncts.Items^[0]);
      lConjuncts.DeleteAll; lConjuncts.Done;
   end
   else lFrm := NewConjFrm(lConjuncts);
case aKind of
   ikTrmFunctor:
      gCorrCond[ord(syCompatibility)] :=
```

```
         NewImpl(lFrm,NewEqFrm(TrmPtr(lId.Pattern),TrmPtr(rId.Pattern)));
      ikFrmPred,ikFrmAttr:
         gCorrCond[ord(syCompatibility)] :=
                  NewImpl(lFrm,NewBicond(FrmPtr(lId.Pattern),FrmPtr(rId.Pattern)));
   end;
   end;
   if not lErrIdentify then
   begin
      CreateLociList(1,dPrimLength,lTypList);
      case aKind of
         ikTrmFunctor:
            begin
               lIdPattern := CopyTerm(TrmPtr(lId.Pattern));
               WithinTerm(TrmPtr(lIdPattern),ChangeToLoci);
               rIdPattern := CopyTerm(TrmPtr(rId.Pattern));
               WithinTerm(TRmPtr(rIdPattern),ChangeToLoci);
            end;
         ikFrmPred,ikFrmAttr:
            begin
               lIdPattern := FrmPtr(lId.Pattern)^.CopyFormula;
               WithinFormula(FrmPtr(lIdPattern),ChangeToLoci);
               rIdPattern := FrmPtr(rId.Pattern)^.CopyFormula;
               WithinFormula(FrmPtr(rIdPattern),ChangeToLoci);
            end;
      end;
      for k := 0 to lEqArgs .Count-1 do  with lEqArgs .Items^[k] do
      begin X := FixedVar[X].nSkelConstNr; Y := FixedVar[Y].nSkelConstNr end;
      lIdentify := new(IdentifyPtr,
                       Init(1+gIdentifications.Count, ArticleID,
                        aKind,lTypList,lIdPattern,rIdPattern,lEqArgs));
//writeln(infofile,'************************************: ', lIdentify^.nConstrKind,CurPos.Line);
//writeln(infofile,'PrimaryList.Count=',lIdentify^.nPrimaryList.Count);
//infotypelist(lIdentify^.nPrimaryList); infonewline;
//infoterm(TrmPtr(lIdentify^.nPattern[0])); infonewline;
//infoterm(TrmPtr(lIdentify^.nPattern[1])); infonewline;
//with lEqArgs do
//for k := 0 to Count-1 do
//writeln(infofile,items^[k].X,'=',items^[k].y);
//infonewline;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_Identify(lIdentify);
      {$ENDIF}
      gIdentifications.Insert(lIdentify);
   end
   else
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elIdentify);
      AReport.Out_XAttr(atAid, ArticleID);
      AReport.Out_XIntAttr(atNr, 0);
      AReport.Out_XAttr(atConstrKind, aKind);
      AReport.Out_XAttrEnd;
      AReport.Out_XEl1(elErrorIdentify);
      AReport.Out_XElEnd(elIdentify);
      {$ENDIF}
   end;
   lId.ArgsSet.Done; rId.ArgsSet.Done;
   ldSet.Done; rdSet.Done;
   lEqArgs.Done;
end;
```

This code is used in chunk 25.
Defines:
DefIdentify, used in chunk 129.
Uses _IdentifyData 56b, Correctness 24, CreateLociList 43c, FindPattern 56b, InitAccess 31c, and SetLociOcc 5c.

62a ⟨*Define property* 62a⟩≡

```
var gPropertyCond: FrmPtr;

procedure DefProperty;
var
   lPropertyNr: integer;
   lType: TypPtr;
   lPos: Position;
   lTypList: MCollection;
   lProperty: PropertyPtr;
begin
   lPropertyNr := InFile.Current.Nr;
   InFile.InPos(CurPos); InFile.InWord;
   lPos := CurPos;
   case PropertyKind(lPropertyNr) of
      sySethood:
         begin
            lType := ReadType;
            CreateLociList(1,dPrimLength,lTypList);
            InitAccess;
            if not AllLociAccessibleInTyp(lTypList,lType) then
            begin
               Error(lPos,100);
            end;
            gPropertyCond := NewExis(NewStandardTyp(ikTypMode,NewEmptyCluster,NewEmptyCluster,
                                                    gBuiltIn[rqSetMode],nil),
                                     NewUniv(lType^.CopyType,
                                             NewPredFrm(ikFrmPred,
                                                        gBuiltIn[rqBelongsTo],
                                                        NewTrmList(NewVarTrm(ikTrmBound,2),
                                                                   NewTrmList(NewVarTrm(ikTrmBound,1),
                                                                              nil)),
                                                        0)));
            lType^.WithinType(ChangeToLoci);
            lProperty := new(PropertyPtr,Init(1+gPropertiesList.Count-RegPropertiesBase, ArticleID,
                                              lTypList,lPropertyNr,lType));
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_PropertyReg(lProperty);
            {$ENDIF}
            gPropertiesList.Insert(lProperty);
         end;
   else
      //   ErrImm(77);
   end;
end;
```

This code is used in chunk 25.
Defines:
DefProperty, used in chunk 129.
gPropertyCond, used in chunk 129.
Uses AllLociAccessibleInTyp 32b, CreateLociList 43c, InitAccess 31c, and ReadType 7b.

62b ⟨*Parse definition of mode — tail* 62b⟩≡

```
// ##TODO: pass patterns too
procedure DefModeTail;
var
   lPattern: PatternPtr;
```

```
begin
   with Notat[noMode], PatternPtr(Items^[Count+fExtCount-1])^ do
                           rConstr.Nr := gWhichOne;
end;
```

This code is used in chunk 25.
Defines:
DefModeTail, used in chunk 121.

63a ⟨*Set structure* 63a⟩≡

```
var gSelectRepresentation: array[1..MaxArgNbr] of FuncTrmPtr;
    gSelectorNr: array[1..MaxArgNbr] of integer;
    gPrimLength: integer;

procedure SetStruct(var fTrm: TrmPtr);
var
   lFuncNr: integer;
   lTrmList: TrmList;
begin
   if fTrm^.TrmSort=ikTrmLocus then
      with VarTrmPtr(fTrm)^ do
         if VarNr > gPrimLength then
            if gSelectRepresentation[VarNr-gPrimLength] = nil then
            begin
               lFuncNr := gSelectorNr[VarNr-gPrimLength];
               dispose(fTrm,Done);
               fTrm := NewLocFuncTrm(ikTrmSelector,lFuncNr,FormalArgs(gPrimLength+1))
            end
            else
            begin
               with gSelectRepresentation[VarNr-gPrimLength]^ do
               begin lFuncNr := FuncNr; lTrmList := CopyTermList(FuncArgs) end;
               dispose(fTrm,Done);
               fTrm := NewLocFuncTrm(ikTrmSelector,lFuncNr,lTrmList);
            end;
end;
```

This code is used in chunk 25.
Defines:
gPrimLength, used in chunk 64.
gSelectorNr, used in chunk 64.
gSelectRepresentation, used in chunk 64.
SetStruct, used in chunk 64.
Uses FormalArgs 6c.

63b ⟨*Analyze selector* 63b⟩≡

```
procedure AnalyzeSelector(var fConstrNr: integer;
                          var fArgList: TrmList;
                          fSelect: integer);
var
   k: integer;
   lTrmList: TrmList;
begin
   lTrmList := fArgList;
   for k := Notat[noSelector].Count-1 downto 0 do
      with PatternPtr(Notat[noSelector].Items^[K])^ do
         if fFormNr=fSelect then
            if CheckTypes(Notat[noSelector].Items^[K],lTrmList) then
            begin
               fArgList := CreateArgList(fPrimTypes.Count);
               RemoveQua(fArgList);
               fConstrNr := rConstr.Nr;
               DisposeListOfTerms(lTrmList);
```

```
            exit;
         end;
      DisposeTrmList(lTrmList);
      fConstrNr := 0;
   end;
```

This code is used in chunk 25.
Defines:
AnalyzeSelector, used in chunk 64.

64 ⟨*Define a structure* 64⟩≡

```
   // ##TODO: this is impossibly long, try some modularisation or clean-up
   // ###TODO: BUG POSSIBLE: I have replaced Prefixes by lPrefixes, it is
   //          theoretically possible that somewhere in this mess WidenningPath
   //          is triggered before the coStructMode is created - check it if problems
   procedure DefStruct;
   var
      j,k,lVarBase,lGenBase,lFieldBase,
      lSelectorNr,llSelectorNr,llPrefNr,lPrefNr,lAbsNr,
      lSelFuncNr,lOldSelNbr,lModeNr: integer;
      lPrefColl: array[1..MaxArgNbr] of NatSet;
      lClusterPtr: AttrCollectionPtr;
      lSelectFuncTyp: array[1..MaxArgNbr] of TypPtr;
      lSelectorTyp,lStructTyp,lTyp,lTyp1:TypPtr;
      lPattern: PatternPtr;
      lStructColl: NatSetPtr;
      lStructSelectors: NatSet;
      lAggrColl: PCollection;
      { -- identyfikacja selektorow -- } { -- selector identification -- }
      lSelectorTerm: SelectorTerm;
      lSelectTyp,llSelectTyp: TypPtr;
      lFuncArgs: TrmList;
      lStructPos: Position;
      lTypList: MCollection;
      lTypPtr: TypPtr;
      r: Integer;
      lPrefixes: MCollection;
      lConstr: ConstrPtr;
      lAbsRegNr: integer;
   label OldSelector;
   begin
      InFile.InPos(CurPos); InFile.InWord;
      { ---- Przeczytanie deklaracji prefiksow ---- }
      { ---- Read prefix declaration ---- }
      gDefPos := CurPos;
      lStructPos := CurPos;
      lOldSelNbr :=  Constr[ coSelector].Count;
      lPrefixes.Init(0,5);
      while InFile.Current.Kind = ikMscPrefix do
      begin
         lTyp := ReadType; Infile.InWord;
         if lTyp^.TypSort = ikTypStruct then
         begin
            if lTyp^.LowerCluster^.Count <> 0 then
               ErrImm(90);
            lPrefixes.Insert(lTyp);
         end
         else ErrImm(errNonStructPrefix);
      end;
      { ======================================== }
      { ---- Wprowadzenie modu strukturowego ---- }
```

```
{ ---- Introduction of structural mode ---- }
gDefPos := CurPos;
GetPattern(noStructMode, lPattern);
CheckAccess(lPattern);
Notat[noStructMode].InsertExt(lPattern);
with Notat[noStructMode], PatternPtr(Items^[Count+fExtCount-1])^.rConstr do
begin Kind := 'L'; Nr :=  Constr[ coStructMode].Count; end;
{ Na zakonczenie typ lokusa odpowiadajacego jedynemu widocznemu
  argumentowi funkcji selektorowych. Jego kopie zostaja zuzyte
  jako
  - typ wynikowy funktora agregujacego, po wstawieniu na pole
  TypeAttributes klastra "abstract",
  - typ podmiotu atrybutu "abstract"
}
{ Finally, the type of locus corresponding to the only visible
  argument of the selector functions. Its copies are consumed as
  - result type of the aggregation functor, after inserting the
    "abstract" cluster into the TypeAttributes field,
  - entity type of the "abstract" attribute
}
lStructTyp := NewStandardTyp(ikTrmAggreg,NewEmptyCluster,NewEmptyCluster,
                             Constr[ coStructMode].Count,FormalArgs(dPrimLength));

{ Zapamietanie sytuacji po deklaracji modu strukturowego }
{ Remembering the situation after the declaration of the structural mode }
gPrimLength := dPrimLength;
lVarBase := g.VarNbr;
lGenBase := g.GenCount;
lFieldBase := gPrimNbr;

{ ---- Przeczytanie lokusow odpowiadajcych polom ---- }
{ ---- Reading locuses corresponding to fields ---- }
GetConstQualifiedList;

{ ---- Wprowadzenie typow funkcji selektorowych ---- }
{ ---- Introduction of selector function types ---- }
for j := 1 to g.VarNbr-lVarBase do
begin
   lSelectFuncTyp[j] := FixedVar[lVarBase+j].nTyp^.CopyType;
   inc(g.GenCount);
   FixedVar[lVarBase+j].nSkelConstNr := g.GenCount;
   LocusAsConst[g.GenCount] := lVarBase+j;
   lSelectFuncTyp[j]^.WithinType(ChangeToLoci);
end;
{ Tutaj jest chyba wykonywana podwojna robota }
{I guess there's double work being done here}
dec(g.GenCount,g.VarNbr-lVarBase);
ParamDecl(lVarBase);

{ ---- Wprowadzenie funktora agregujacego ---- }
{ ---- Introduction of the aggregation functor ---- }
gDefPos := CurPos;
lPattern := new(PatternPtr,Init(noAggregate, AbsNotatNr(noAggregate),
                                ArticleID));
Notat[noAggregate].InsertExt(lPattern);

InitAccess;
{ Specjalna realizacja GetFormat }
{ Special implementation of GetFormat }
for j := lFieldBase+1 to gPrimNbr do LociOcc[j] := true;
```

```
with Notat[noAggregate], PatternPtr(Items^[Count+fExtCount-1])^ do
begin
   { Przewiniecie formatu konstruktora }
   { Constructor Format Scroll }
   fFormNr := InFile.Current.Nr; InFile.InPos(CurPos); InFile.InWord;
   { Inicjalizacja } { Initialization }
   fPrimTypes.Init(dPrimLength);
   for k := 1 to dPrimLength do
      fPrimTypes.Insert(gPrimaries[k]^.CopyType);
   Visible.Init(dPrimLength-gPrimLength);
   for k := gPrimLength+1 to dPrimLength do r := Visible.Insert(k);
   CheckAccess(Items^[Count+fExtCount-1]);
   if PatternPtr(Notat[noStructMode].Items^[Notat[noStructMode].Count+
                                            Notat[noStructMode].fExtCount-1])^.fFormNr = 0 then
      fFormNr := 0;
   with rConstr do
   begin Kind := ikTrmAggreg; Nr :=  Constr[ coAggregate].Count; end;
   lTypPtr := lStructTyp^.CopyType;
   lAbsNr :=  1 + Constr[coAggregate].Count - ConstrBase[coAggregate];
   lConstr :=  new(AggrConstrPtr,
                  InitForPattern(lAbsNr,ArticleID,fPrimTypes,lTypPtr));
   AggrConstrPtr(lConstr)^.fAggregBase :=  gPrimLength;
   Constr[ coAggregate].Insert(lConstr);
   // ##NOTE: fAggrColl is done later
end;

{ ======================================== }
{ Przywrocenie sytuacji po deklaracji modu strukturowego }
{ Restoring the situation after declaration of a structured module }
RenewPrimaries(gPrimLength);
for j := lVarBase+1 to g.VarNbr do dispose(FixedVar[j].nTyp,Done);
g.VarNbr := lVarBase; g.GenCount := lGenBase;

{ Wprowadzamy wspolny lokus dla atrybutu "abstract"
  i dla funkcji selektorowych }
{ We introduce a common locus for the "abstract"
  attribute and for selector functions }
AppendLocus(lStructTyp);

{ --- Inicjalizacja kolekcji selektorow ---- }
{ --- Initialization of selector collection ---- }
lAggrColl := new(PCollection, Init(2,2));
lStructColl := new(NatSetPtr, Init(2,2));

{ ---- Inicjalizacja prefiksow ---- }
{ ---- Prefix initialization ---- }
for lPrefNr := 0 to lPrefixes.Count-1 do
begin
   lModeNr :=  TypPtr(lPrefixes.Items^[lPrefNr])^.ModNr;
   with StructConstrPtr(Constr[ coStructMode].At(lModeNr))^ do
      lPrefColl[lPrefNr+1].CopyNatSet(fFields^);
end;

{ ---- Wprowadzenie funktora zapominania ---- }
{ ---- Introduction of the forgetting functor ---- }
gDefPos := CurPos;
lPattern := new(PatternPtr,Init(noForgetFunctor, AbsNotatNr(noForgetFunctor),
                                ArticleID));
Notat[noForgetFunctor].InsertExt(lPattern);
with Notat[noForgetFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
```

```
begin
   fFormNr := InFile.Current.Nr; InFile.InPos(CurPos); InFile.InWord;
   fPrimTypes.Init(dPrimLength);
   for k := 1 to dPrimLength do
      fPrimTypes.Insert(gPrimaries[k]^.CopyType);
   Visible.Init(1); r := Visible.Insert(dPrimLength);
   rConstr.Kind := ikTrmSubAgreg;
   rConstr.Nr :=  Constr[ coStructMode].Count;
end;

{ ---- Wprowadzenie funktorow selektorowych ---- }
{ ---- Introduction of selector functors ---- }
lSelFuncNr := 0;
fillchar(gSelectRepresentation,SizeOf(gSelectRepresentation),0);
lStructSelectors.Init(MaxArgNbr,MaxArgNbr);
while InFile.Current.Kind=ikTrmSelector do
begin
   GetPattern(noSelector, lPattern);
   if lStructSelectors.ElemNr(lPattern^.fFormNr) >= 0 then
      ErrImm(errFieldHomonimy);
   lStructSelectors.InsertElem(lPattern^.fFormNr);
   CheckAccess(lPattern);
   InFile.InWord;
   inc(lSelFuncNr); lSelectorTyp := lSelectFuncTyp[lSelFuncNr];
   lSelectorTyp^.WithinType(SetStruct);
   for lPrefNr := 0 to lPrefixes.Count-1 do
   begin
      FixedVar[g.VarNbr].nTyp := TypPtr(lPrefixes.Items^[lPrefNr])^.CopyType;
      lSelectorTerm.Init(lPattern^.fFormNr,CurPos,nil);
      lFuncArgs := NewTrmList(NewVarTrm(ikTrmConstant,g.VarNbr),nil);
      AnalyzeSelector(lSelectorNr,lFuncArgs,lSelectorTerm.Select);
      dispose(FixedVar[g.VarNbr].nTyp,Done);
      if lSelectorNr <> 0 then
      begin
         gSelectRepresentation[lSelFuncNr] := NewLocFuncTrm(ikTrmSelector,lSelectorNr,lFuncArgs);
         lSelectTyp := ConstrTypPtr(Constr[coSelector].Items^[lSelectorNr]
                                 )^.fConstrTyp^.InstTyp(lFuncArgs);
         gSuperfluous := 0;
         lSelectorTyp^.WithinType(ChangeToConst);
         if not EqTyp(lSelectorTyp,lSelectTyp) then
            ErrImm(errFieldTypeInconsistent);
         dispose(lSelectTyp,Done);
         for llPrefNr := lPrefNr+1 to lPrefixes.Count-1 do
         begin
            inc(g.VarNbr);
            FixedVar[g.VarNbr].nIdent := 0;
            FixedVar[g.VarNbr].nTyp := TypPtr(lPrefixes.Items^[llPrefNr])^.CopyType;
            lSelectorTerm.Init(lPattern^.fFormNr,CurPos,nil);
            lFuncArgs := NewTrmList(NewVarTrm(ikTrmConstant,g.VarNbr),nil);
            AnalyzeSelector(llSelectorNr,lFuncArgs,lSelectorTerm.Select);
            dispose(FixedVar[g.VarNbr].nTyp,Done);
            dec(g.VarNbr);
            if llSelectorNr <> 0 then
            begin
               if lSelectorNr = llSelectorNr then
               begin
                  llSelectTyp := ConstrTypPtr(Constr[coSelector].Items^[lSelectorNr]
                                           )^.fConstrTyp^.InstTyp(lFuncArgs);
                  llSelectTyp^.WithinType(RenewConst);
                  if not EqTyp(lSelectorTyp,llSelectTyp) then
```

```
                    ErrImm(errFieldTypeInconsistent);
                 dispose(llSelectTyp,Done);
                 lPrefColl[llPrefNr+1].DeleteElem(llSelectorNr)
              end
              else ErrImm(errFieldHomonimy);
              DisposeTrmList(lFuncArgs);
           end;
        end;
        lPrefColl[lPrefNr+1].DeleteElem(lSelectorNr);
        dispose(lSelectorTyp,Done);
        lPattern^.Visible.Done;
        goto OldSelector;
     end;
  end;
  { -- Jest to nowy selektor -- }
  { -- This is a new selector -- }
  with lPattern^.rConstr do
  begin Kind := ikTrmSelector; Nr :=  Constr[ coSelector].Count; end;
  Notat[noSelector].InsertExt(lPattern);
  lAbsNr :=  1 + Constr[coSelector].Count - ConstrBase[coSelector];
  with lPattern^ do
     lConstr :=  new(ConstrTypPtr,
                     InitForPattern(coSelector,lAbsNr,ArticleID,
                                    fPrimTypes,lSelectorTyp));
  Constr[ coSelector].Insert(lConstr);

  dispose(lSelectorTyp,Done);
  lSelectorNr :=  Constr[ coSelector].Count - 1;
  OldSelector:
     { -- Zapisanie przekodowania lokusa na selektor, dla SetStruct -- }
     { -- Saving the locus recoding to a selector, for SetStruct -- }
     gSelectorNr[lSelFuncNr] := lSelectorNr;
  { -- Wstawienie selektora do kolekcji -- }
  { -- Inserting a selector into the collection -- }
  lStructColl^.InsertElem(lSelectorNr);
  lAggrColl^.Insert(new(PIntItem,Init(lSelectorNr)));
end;
lStructSelectors.Done;

for j := 1 to lSelFuncNr do
  if gSelectRepresentation[j] <> nil then
     DisposeTrm(gSelectRepresentation[j]);

{ ---- Sprawdzamy czy prefiksy zostaly wyczerpane ---- }
{ ---- We check whether the prefixes have been exhausted ---- }
for lPrefNr := 0 to lPrefixes.Count-1 do
begin
  if lPrefColl[lPrefNr+1].Count <> 0 then ErrImm(errIncompletePrefix);
  TypPtr(lPrefixes.Items^[lPrefNr])^.WithinType(ChangeToLoci);
  lPrefColl[lPrefNr+1].Done;
end;

{ ---- Wstawienie kolekcji selektorow ---- }
{ ---- Inserting a collection of selectors ---- }
with AggrConstrPtr(Constr[coAggregate].Last)^ do
  fAggrColl :=  lAggrColl;
lAbsNr :=  1 + Constr[coStructMode].Count - ConstrBase[coStructMode];
with Notat[noStructMode], PatternPtr(Items^[Count+fExtCount-1])^ do
lConstr := new(StructConstrPtr,
               InitForPattern(lAbsNr,ArticleID,fPrimTypes));
```

```
with StructConstrPtr(lConstr)^ do
begin
   fFields := lStructColl;
   fStructModeAggrNr := Constr[coAggregate].Count - 1;
   fPrefixes.MoveCollection(lPrefixes);
end;
Constr[ coStructMode].Insert(lConstr);

{ ---- Wprowadzenie atrybutu "abstract" ---- }
{ ---- Entering the "abstract" attribute ---- }
gDefPos := lStructPos;
lPattern := new(PatternPtr,Init(noAttribute, AbsNotatNr(noAttribute),
                                ArticleID));
Notat[noAttribute].InsertExt(lPattern);
lTypPtr := lStructTyp^.CopyType;
lAbsNr :=  1 + Constr[coAttribute].Count - ConstrBase[coAttribute];
with Notat[noAttribute], PatternPtr(Items^[Count+fExtCount-1])^ do
begin
   fFormNr := 1 { "abstract' };
   fAntonymic := false;
   {??} {Co to za numer formatu, czemu nie stala z BuitIn ?}
   {What format number is this, why wasn't it from BuitIn?}
   fPrimTypes.Init(dPrimLength);
   for k := 1 to dPrimLength do
      fPrimTypes.Insert(gPrimaries[k]^.CopyType);
   Visible.Init(1);
   r := Visible.Insert(FixedVar[g.VarNbr].nSkelConstNr);
   rConstr.Kind := ikFrmAttr; rConstr.Nr := Constr[coAttribute].Count;
   lConstr :=  new(ConstrTypPtr, InitForPattern(coAttribute,lAbsNr,ArticleID,
                                                fPrimTypes,lTypPtr));
   include(lConstr^.fProperties, syAbstractness);
   Constr[ coAttribute].Insert(lConstr);
end;

// Register the existential cluster
lClusterPtr := new(AttrCollectionPtr,Init(2,4));
{??} {czemu tutaj wystepuje 1, czy nil jest poprawne?}
{??} {why is there 1 here, is nil correct?}
lClusterPtr^.InsertAttr(Constr[coAttribute].Count-1, 1,
                        FormalArgs(dPrimLength-1){###});
// lClusterPtr^.WithinAttrCollection(ChangeToLoci);
{ Nie potrzeba sprawdzac, bo jest jednoelementowy }
{ No need to check because it is single-element }
CreateLociList(1,dPrimLength-1,lTypList);
lTyp1 := lTypPtr^.CopyType;
lAbsRegNr :=  1 + RegisteredCluster.Count +  RegisteredCluster.fExtCount - RegClusterBase;
RegisteredCluster.InsertExt(new(RClusterPtr,
                                RegisterCluster(lAbsRegNr,ArticleID,lClusterPtr,lTypList,lTyp1)));
dispose(lTyp1,Done);
with  AggrConstrPtr(Constr[coAggregate].Last)^.fConstrTyp^ do
begin
   dispose(LowerCluster,Done);
   LowerCluster := {CopyCluster(}lClusterPtr{)};
   dispose(UpperCluster,Done);
   UpperCluster := CopyCluster(LowerCluster);
end;
{ Przywrocenie sytuacji po deklaracji modu strukturowego }
{ Restoring the situation after declaration of a structured module }
RenewPrimaries(gPrimLength);
g.VarNbr := lVarBase;
```

```
    g.GenCount := lGenBase;
    dispose(lStructTyp,Done);

    // ##NOTE: the attribute has to go first, or probably at least
    //         before the coAggregate. It has to be know before any
    //         cluster containing it is read, otherwise the CompAttr
    //         comparison function used for clusters causes internal error.
    //         This is fairly fragile, if their were more such mutually
    //         defined constructors we could get into serious trouble.
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_Constructor(Constr[ coAttribute].Last,
                            Constr[coAttribute].Count - 1);
    AReport.Out_Constructor(Constr[ coStructMode].Last,
                            Constr[coStructMode].Count - 1);
    AReport.Out_Constructor(Constr[ coAggregate].Last,
                            Constr[coAggregate].Count - 1);

    with Constr[ coSelector] do
       for k :=  lOldSelNbr to Count-1 do
          AReport.Out_Constructor(Items^[ k], k);

    AReport.Out_XElStart0(elRegistration);
    with RegisteredCluster do
       AReport.Out_RCluster(Items^[Count+fExtCount-1]);
    AReport.Out_XElEnd(elRegistration);
    {$ENDIF}
  end;
```

This code is used in chunk 25.
Defines:
DefStruct, used in chunk 121.
Uses AbsNotatNr 31a, AnalyzeSelector 63b, AppendLocus 15a, ChangeToConst 6a, CheckAccess 31d, CreateLociList 43c, FormalArgs 6c, GetConstQualifiedList 14a, GetPattern 31b, gPrimLength 63a, gSelectorNr 63a, gSelectRepresentation 63a, InitAccess 31c, ParamDecl 15b, ReadType 7b, RenewConst 6b, RenewPrimaries 5b, and SetStruct 63a.

70 ⟨*Parse a reservation* 70⟩≡

```
  procedure Reservation;
  var
     lExpPtr: ExpPtr;
     k: integer;
     lIdents: IntSequence;
  begin
     BoundVarNbr := 0;
     {$IFDEF ANALYZER_REPORT}
     AReport.Out_XElStart0(elReservation);
     {$ENDIF}
     {----}
     MarkTermsInTTColl;
     {----}
     { czytanie typow zmiennych wolnych w typie rezerwacji }
     { reading the types of free variables in the reservation type }
     gExportableItem := true;
     gConstInExportableItemOcc := false;
     lIdents.Init(0);
     InFile.InWord;
     while InFile.Current.Kind <> ';' do
     begin
        lIdents.Insert(InFile.Current.Nr);
        InFile.InWord;
     end;
     {$IFDEF ANALYZER_REPORT}
     for k  :=  0 to lIdents.fCount - 1 do
```

```
    begin
       AReport.Out_XElStart(elIdent);
       AReport.Out_XIntAttr(atVid, lIdents.fList^[k]);
       AReport.Out_XElEnd0;
    end;
    {$ENDIF}
    InFile.InWord;
    while InFile.Current.Kind <> ';' do
    begin
       lExpPtr := LoadType;
       inc(BoundVarNbr);
       BoundVar[BoundVarNbr] := lExpPtr^.Analyze;
       dispose(lExpPtr,Done);
       InFile.InWord;
    end;
    inc(ResNbr);
    mizassert(2524,ResNbr<=MaxResNbr);
    lExpPtr := LoadType;
    ReservedVar[ResNbr] := lExpPtr^.Analyze;
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_Type(ReservedVar[ResNbr]);
    {$ENDIF}
    dispose(lExpPtr,Done);
    {----}
    RemoveTermsFromTTColl;
    {----}
    for k := 1 to BoundVarNbr do dispose(BoundVar[k],Done);
    Infile.InWord;
    BoundVarNbr := 0;
    gExportableItem := false;
    gConstInExportableItemOcc := false;
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_XElEnd(elReservation);
    {$ENDIF}
  end;
```

This code is used in chunk 25.
Defines:
  Reservation, used in chunk 138.
Uses Analyze 138.

71 ⟨*Spread local predicates* 71⟩≡

```
  procedure RegularStatement(var fFrm: FrmPtr); FORWARD;

  procedure SpreadLocPred(var aFrm: FrmPtr);
  var
     lFrm,lFrm1,lFrm2: FrmPtr;
  begin
     repeat
        while aFrm^.FrmSort=ikFrmPrivPred do
        begin
           lFrm := aFrm;
           aFrm := LocPredFrmPtr(aFrm)^.PredExp;
           DisposeTrmList(LocPredFrmPtr(lFrm)^.PredArgs);
           dispose(lFrm);
        end;
        if (aFrm^.FrmSort=ikFrmNeg) and (NegFrmPtr(aFrm)^.NegArg^.FrmSort = ikFrmPrivPred) then
        begin
           lFrm1 := LocPredFrmPtr(NegFrmPtr(aFrm)^.NegArg)^.PredExp;
           while lFrm1^.FrmSort=ikFrmPrivPred do
              lFrm1 := LocPredFrmPtr(lFrm1)^.PredExp;
           if lFrm1^.FrmSort = ikFrmNeg then
```

```
            begin
               lFrm2 := aFrm;
               aFrm := NegFrmPtr(lFrm1)^.NegArg;
               lFrm := NegFrmPtr(lFrm2)^.NegArg;
               dispose(lFrm2);
               while lFrm^.FrmSort=ikFrmPrivPred do
               begin
                  lFrm2 := lFrm;
                  lFrm := LocPredFrmPtr(lFrm)^.PredExp;
                  DisposeTrmList(LocPredFrmPtr(lFrm2)^.PredArgs);
                  dispose(lFrm2);
               end;
               dispose(lFrm1);
            end;
         end
      until aFrm^.FrmSort<>ikFrmPrivPred;
   end;
```

This code is used in chunk 25.
Defines:
SpreadLocPred, used in chunks 72b, 73, and 75c.
Uses RegularStatement 97.

72a ⟨*Decompose formula* 72a⟩≡

```
   // If conjunction head is the first conjunct, tail is the rest.
   // Otherwise tail is verum and head is the whole fla.
   procedure Decompose(fFrm: FrmPtr; var fFrm_head,fFrm_tail: FrmPtr);
   { zakladamy, ze "decompose" dziala na kopii i moze ja niszczyc }
   { we assume that "decompose" works on the copy and may destroy it }
   begin
      if fFrm^.FrmSort = ikFrmConj then
         with ConjFrmPtr(fFrm)^ do
      begin
         fFrm_head := FrmPtr(Conjuncts.Items^[0]);
         Conjuncts.AtDelete(0);
         if Conjuncts.Count = 1 then
         begin fFrm_Tail := FrmPtr(Conjuncts.Items^[1]);
         Conjuncts.DeleteAll; dispose(fFrm,Done);
         end
         else fFrm_tail := fFrm;
      end
      else begin fFrm_head := fFrm; fFrm_tail := NewVerum end;
   end;
```

This code is used in chunk 25.
Defines:
Decompose, used in chunks 73, 82, and 85.

72b ⟨*Spread atomic formula* 72b⟩≡

```
   // ##NOTE: destructive on fFrm, usually requires a copy
   // Tries to replace atomic fFrm with its definitional expansion,
   // starting from the most recent items in Definientia.
   // If no success, replaces fFrm with NewInCorFrm.
   // This now returns the number of the Definientia item, or -1 if none.
   function SpreadAtomicFormula(var fFrm: FrmPtr; Conclusion: boolean): integer;
   var
      lArgs,lElem: TrmList;
      lWord: Lexem;
      i,lResult: integer;
      Negated: boolean;
      lItem: DefiniensPtr;
      lFrm: FrmPtr;
   begin
```

```
      lResult :=  -1;
      Negated := false;
      SpreadLocPred(fFrm);
      if fFrm^.FrmSort=ikFrmNeg then
      begin
         lFrm := fFrm;
         fFrm := NegFrmPtr(fFrm)^.NegArg;
         SpreadLocPred(fFrm);
         dispose(lFrm);
         Negated := true
      end;
      lWord.Kind := fFrm^.FrmSort;
      lItem := nil;
      case lWord.Kind of
         ikFrmPred: AdjustFrm(PredFrmPtr(fFrm),lWord.Nr,lArgs);
         ikFrmAttr: AdjustAttrFrm(PredFrmPtr(fFrm),lWord.Nr,lArgs);
         ikFrmQual:
            with QualFrmPtr(fFrm)^,QualTyp^ do
            if (TypSort = ikTypMode) and (LowerCluster^.Count = 0) then
            begin
               lWord.Kind := ikTypMode;
               QualTyp^.AdjustTyp(lWord.Nr,lArgs);
               lElem := nil;
               if lArgs=nil then lArgs := NewTrmList(QualTrm,nil)
               else
               begin
                  lElem := LastElem(lArgs);
                  lElem^.NextTrm := NewTrmList(QualTrm,nil);
               end;
            end;
      end;
      lItem := nil;
      for i := Definientia.Count-1 downto 0 do
         if Matches(lWord,lArgs,DefiniensPtr(Definientia.Items^[i])) then
         begin lItem := Definientia.Items^[i]; lResult :=  i + 1; break end;
      if lWord.Kind = ikTypMode then
         if lElem <> nil then
         begin
            dispose(lElem^.NextTrm);
            lElem ^.NextTrm := nil;
         end
         else dispose(lArgs);
      dispose(fFrm,Done);
      SpreadAtomicFormula :=  lResult;
      if lItem = nil then begin fFrm := NewInCorFrm; exit end;
      fFrm := lItem^.SpreadFrm(CreateArgList(lItem^.PrimaryList.Count),Negated,Conclusion);
   end;
```

This code is used in chunk 25.
Defines:
SpreadAtomicFormula, used in chunks 73, 75c, 82, and 85.
Uses SpreadLocPred 71.

73 ⟨*Chopping definientia(?)* 73⟩≡

```
   const MaxExpansionNbr = 20;

   // ##NOTE: destructive on fForm, usually requires a copy
   // ##NOTE: seems also destructive on g.Thesis, so failure
   //         probably means that g.Thesis is messed up
   // ##TODO: a version using a second formula rather than g.Thesis
   //         would be much cleaner and safer
   // In fDefs we pass the numbers of items in Definientia, that were
```

```
// succesfully used for chopping, together with their counts.
function Chopped(fForm: FrmPtr;
                 Conclusion: boolean;
                 var fDefs: NatFuncPtr): boolean;
var
   f_head,Thesis_head: FrmPtr;
   ii,lDefNr: integer;
label ToChop;
procedure DisposeInLoop;
begin
   dispose(Thesis_head, Done);
   dispose(f_head, Done);
   dispose(fForm, Done);
   fDefs^.DeleteAll;
end;
begin
   fDefs :=  new(NatFuncPtr, InitNatFunc(4,4));
   Chopped := true; if g.Thesis^.FrmSort=ikError then exit;
   if fForm^.FrmSort=ikError then begin g.Thesis := NewInCorFrm; exit end;
   // This loop ends when there is no more conjunct in fForm
   while fForm^.FrmSort<>ikFrmVerum do
   begin
      Decompose(fForm,f_head,fForm);
      Decompose(g.Thesis,Thesis_head,g.Thesis);
      // Now we try to spread (apply definiens to) the Thesis_head
      // until it is equal to f_head. If no success, we exit with false.
      for ii := 1 to MaxExpansionNbr do
      begin
         if EqFrm(f_head,Thesis_head) then goto ToChop;
         repeat
            if (f_head^.FrmSort = ikFrmPrivPred) and (Thesis_head^.FrmSort = ikFrmPrivPred) then
            begin
               case CompareInt(LocPredFrmPtr(f_head)^.PredNr, LocPredFrmPtr(Thesis_head)^.PredNr) of
                  -1:
                     begin
                        SpreadLocPred(Thesis_head);
                        Decompose(NewConj(Thesis_head,g.Thesis),Thesis_head,g.Thesis);
                     end;
                  0: begin DisposeInLoop; Chopped := false; exit; end;
                  1:
                     begin
                        SpreadLocPred(f_head);
                        Decompose(NewConj(f_head,fForm),f_head,fForm);
                     end;
               end;
            end;
            if EqFrm(f_head,Thesis_head) then goto ToChop;
            SpreadLocPred(f_head);
            Decompose(NewConj(f_head,fForm),f_head,fForm);
            if EqFrm(f_head,Thesis_head) then goto ToChop;
            SpreadLocPred(Thesis_head);
            Decompose(NewConj(Thesis_head,g.Thesis),Thesis_head,g.Thesis);
            if EqFrm(f_head,Thesis_head) then goto ToChop;
         until (f_head^.FrmSort <> ikFrmPrivPred) and (Thesis_head^.FrmSort <> ikFrmPrivPred);
         lDefNr :=  SpreadAtomicFormula(Thesis_head,Conclusion);
         if lDefNr >= 0 then fDefs^.Up(lDefNr);
         if Thesis_head^.FrmSort = ikError then
         begin DisposeInLoop; Chopped := false; exit; end;
         Decompose(NewConj(Thesis_head,g.Thesis),Thesis_head,g.Thesis);
      end; // of the for loop
```

```
        { Tutaj by tez trzeba dysponowac !} {You would also need to have it here!}
        Chopped := false; fDefs^.DeleteAll; exit;
        ToChop: // success for heads
           dispose(Thesis_head,Done);
        dispose(f_head,Done);
     end; // of the while loop
     dispose(fForm);
  end;
```

This code is used in chunk 25.
Defines:
  Chopped, used in chunks 77b, 78a, and 82.
  MaxExpansionNbr, used in chunks 75c and 85.
Uses Decompose 72a, SpreadAtomicFormula 72b, and SpreadLocPred 71.

75a ⟨*Change bound variable to declaration(?)* 75a⟩≡

```
  procedure ChangeBoundToDecl(var fTrm: TrmPtr);
  begin
    with VarTrmPtr(fTrm)^ do
       if TrmSort=ikTrmBound then
          if VarNr>1 then dec(VarNr)
          else begin TrmSort := ikTrmConstant; inc(VarNr,gFixedBase) end;
  end;
```

This code is used in chunk 25.
Defines:
  ChangeBoundToDecl, used in chunk 75c.
Uses gFixedBase 12c.

75b ⟨*Mark term as taken* 75b⟩≡

```
  procedure SetTaken(var fTrm: TrmPtr);
  begin
     with VarTrmPtr(fTrm)^ do
        if (TrmSort=ikTrmConstant) and (VarNr=g.VarNbr) then
        begin TrmSort := ikTrmLocus; VarNr := 1 end;
  end;
```

This code is used in chunk 25.
Defines:
  SetTaken, used in chunk 78c.

75c ⟨*Chop variables* 75c⟩≡

```
  // ###TODO: the parts creating implications should be removed,
  //          check how often it is used in MML, in case of multiple
  //          variables it behaves very strangely
  // ##NOTE: seems destructive on g.Thesis, so failure
  //         probably means that g.Thesis is messed up
  // ##TODO: a version using a second formula rather than g.Thesis
  //         would be much cleaner and safer
  // In the result we pass the numbers of items in Definientia, that were
  // succesfully used for chopping, together with their counts.
  function ChopVars(fWidenable,Conclusion: boolean;
                    fPos: Position): NatFuncPtr;
  var
     ii,kk,lDefNr: integer;
     lTh: FrmPtr;
     lTyp,lTyp1: TypPtr;
     lDefs: NatFuncPtr;
  label ToChop;
  begin
     lDefs := new(NatFuncPtr, InitNatFunc(4,4));
     ChopVars := lDefs;
     if g.Thesis^.FrmSort=ikError then exit;
     for kk := gFixedBase+1 to g.VarNbr do
```

```
begin
   // expand definientia until the thesis is UnivFrm
   for ii := 1 to MaxExpansionNbr do
   begin
      SpreadLocPred(g.Thesis);
      if g.Thesis^.FrmSort = ikFrmUniv then goto ToChop;
      lDefNr := SpreadAtomicFormula(g.Thesis,Conclusion);
      if lDefNr >= 0 then lDefs^.Up(lDefNr);
      if g.Thesis^.FrmSort = ikError then
      begin Error(fPos,55); lDefs^.DeleteAll; exit end;
   end;
   Error(fPos,55); g.Thesis := NewInCorFrm; lDefs^.DeleteAll; exit;
   ToChop:
      WithinFormula(g.Thesis,ChangeBoundToDecl); inc(gFixedBase);
      with g.Thesis^ do
   begin
      if FixedVar[kk].nTyp^.TypSort=ikError then
      begin g.Thesis := NewInCorFrm; lDefs^.DeleteAll; exit end;
      if fWidenable then
      begin lTyp := UnivFrmPtr(g.Thesis)^.Quantified^.CopyType;
      { Argument IsWiderThan jest rozdysponowywany. }
      { The IsWiderThan argument is distributed. }
      if not UnivFrmPtr(g.Thesis)^.Quantified^.IsWiderThan(FixedVar[kk].nTyp^.CopyType) then
      begin
         if not FixedVar[kk].nTyp^.IsWiderThan(lTyp^.CopyType)
            or not EqualClusters(FixedVar[kk].nTyp,lTyp,EqAttr) then
         begin Error(fPos,57); g.Thesis := NewInCorFrm; lDefs^.DeleteAll; exit end;
         lTh := UnivFrmPtr(g.Thesis)^.Scope^.CopyFormula;
         dispose(g.Thesis,Done);
         g.Thesis := lTh;
         repeat
            g.Thesis :=
               NewImpl(NewQualFrm(NewVarTrm(ikTrmConstant,kk),lTyp^.CopyType),g.Thesis);
            lTyp1 := lTyp^.Widening;
            dispose(lTyp,Done);
            lTyp := lTyp1;
         until lTyp^.EqRadices(FixedVar[kk].nTyp);
         dispose(lTyp,Done); exit;
      end;
      dispose(lTyp,Done);
      end
      else if not EqTyp(UnivFrmPtr(g.Thesis)^.Quantified,FixedVar[kk].nTyp) then
      begin
         lTyp := UnivFrmPtr(g.Thesis)^.Quantified^.CopyType;
         if not FixedVar[kk].nTyp^.IsWiderThan(lTyp^.CopyType)
            or not EqualClusters(FixedVar[kk].nTyp,lTyp,EqAttr) then
         begin ErrImm(56); g.Thesis := NewInCorFrm; lDefs^.DeleteAll; exit end;
         lTh := UnivFrmPtr(g.Thesis)^.Scope^.CopyFormula;
         dispose(g.Thesis,Done);
         g.Thesis := lTh;
         repeat
            g.Thesis :=
               NewImpl(NewQualFrm(NewVarTrm(ikTrmConstant,kk),lTyp^.CopyType),
                       g.Thesis);
            lTyp1 := lTyp^.Widening;
            dispose(lTyp,Done);
            lTyp := lTyp1;
            if lTyp = nil then exit;
         until lTyp^.EqRadices(FixedVar[kk].nTyp);
         dispose(lTyp,Done);
```

```
            exit;
         end;
         lTh := g.Thesis;
         g.Thesis := UnivFrmPtr(g.Thesis)^.Scope;
         UnivFrmPtr(lTh)^.Scope := NewVerum; dispose(lTh,Done);
      end;
   end;
end;
```

This code is used in chunk 25.
Defines:
ChopVars, used in chunk 78c.
Uses ChangeBoundToDecl 75a, gFixedBase 12c, MaxExpansionNbr 73, SpreadAtomicFormula 72b, and SpreadLocPred 71.

77a ⟨*Is* Position *in the collection?* 77a⟩≡

```
{$IFDEF SKLTTEST}
function PosInCollection(fPos : Position): boolean;
var
   res  : boolean;
   i    : integer;
   lPos : PPosition;
begin
   res := false;
   with gThesisPosCollection do
      for i := Count-1 downto 0 do
      begin
            lPos := At(i);
            if (lPos^.Pos.Line = fPos.Line) and (lPos^.Pos.Col = fPos.Col) then
            begin
               res := true;
               break;
            end;
      end;
   PosInCollection := res;
end;
{$ENDIF}
```

This code is used in chunk 25.
Defines:
PosInCollection, used in chunk 77b.

77b ⟨*Chop conclusion* 77b⟩≡

```
function ChopConcl(fForm: FrmPtr; fPos: Position): NatFuncPtr;
var
   lDefs: NatFuncPtr;
begin
   {$IFDEF SKLTTEST}
   {$IFDEF MDEBUG}
   if fForm^.FrmSort in [ikFrmConj, ikFrmNeg, ikFrmPred] then
      writeln('w ChopConcl=',fForm^.FrmSort);
   {$ENDIF}
   if not PosInCollection(fPos) then
      if fForm^.FrmSort <> ikFrmPred then
            WrongSkeleton('Too complex conclusion',fPos);
   {$ENDIF}
   if not Chopped(fForm,true,lDefs) then
   begin g.Thesis := NewIncorFrm; Error(fPos,51) end;
   ChopConcl := lDefs;
end;
```

This code is used in chunk 25.
Defines:
ChopConcl, used in chunk 78c.
Uses Chopped 73 and PosInCollection 77a.

78a ⟨*Chop assumption* 78a⟩≡

```
function ChopAssum(fForm: FrmPtr; fPos: Position): NatFuncPtr;
var
   lDefs: NatFuncPtr;
begin
   {$IFDEF SKLTTEST}
   {$IFDEF MDEBUG}
   if fForm^.FrmSort in [ikFrmConj, ikFrmNeg, ikFrmPred] then
      writeln('w ChopAssume=',fForm^.FrmSort);
   {$ENDIF}
   {$ENDIF}
   g.Thesis := NewNegDis(g.Thesis);
   if not Chopped(fForm,false,lDefs) then
   begin g.Thesis := NewIncorFrm; Error(fPos,52);
   end;
   g.Thesis := NewNegDis(g.Thesis);
   ChopAssum := lDefs;
end;
```

Root chunk (not used in this document).
Defines:
ChopAssum, used in chunk 78c.
Uses Chopped 73.

78b ⟨*Dispose level* 78b⟩≡

```
procedure DisposeLevel(const f: LevelRec);
var
   i: integer;
begin
   for i := f.VarNbr+1 to g.VarNbr do
   begin
      if FixedVar[i].nExp then DisposeTrm(FixedVar[i].nDef);
      dispose(FixedVar[i].nTyp,Done);
   end;
   LocPredDef.FreeItemsFrom(f.LocPredNbr);
   LocFuncDef.FreeItemsFrom(f.LocFuncNbr);
   g := f;
   {----}
   RemoveTermsFromTTColl;
   {----}
end;
```

This code is used in chunk 25.
Defines:
DisposeLevel, used in chunks 82, 85, 92, 95, 121, 129, 132, and 137b.

78c ⟨*Reasoning* 78c⟩≡

```
procedure Statement; forward;

procedure HereBy(var fFrm: FrmPtr); forward;

// This is used when thesis is known
procedure Reasoning;
var
   lVarBase,i,lId: integer;
   lTrm: TrmPtr;
   lPos: Position;
   lConditions: MCollection;
   lFrm: FrmPtr;
   lTyp: TypPtr;
   lTrmList: TrmList;
   {$IFDEF SKLTTEST} ww: Integer; {$ENDIF}
   lDefs: NatFuncPtr;
```

```
procedure WriteThesis;
begin
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart0(elThesis);
   AReport.Out_Formula(g.Thesis);
   AReport.Out_NatFunc(elThesisExpansions, lDefs^);
   AReport.Out_XElEnd(elThesis);
   {$ENDIF}
   dispose(lDefs, Done); lDefs := nil;
end;
begin
   while InFile.Current.Kind <>ikMscEndBlock do
   begin
      case InFile.Current.Kind of
         ikItmGeneralization:
            begin
               InFile.InPos(lPos);
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XElStart(elLet);
               AReport.Out_XIntAttr(atNr, g.VarNbr+1);
               AReport.Out_XAttrEnd;
               {$ENDIF}
               if g.Thesis^.FrmSort = ikFrmFlexConj then
                  g.Thesis := FlexFrmPtr(g.Thesis)^.nExpansion;
               GetQualifiedList;
               for i := gFixedBase+1 to g.VarNbr do
               begin
                  inc(g.GenCount); { trzeba spradzic, czy to potrzebne }
                  {you need to check if it is necessary}
                  FixedVar[i].nSkelConstNr := g.GenCount;
               end;
               WriteQualified;
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XElEnd(elLet);
               {$ENDIF}
               lDefs := ChopVars(false,false,lPos);
               WriteThesis;
            end;
         ikItmAssumption:
            begin
               InFile.InPos(lPos);
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XElStart0(elAssume);
               {$ENDIF}
               InFile.InWord;
               ReadPropositions(lConditions);
               {$IFDEF SKLTTEST} {assumption}
                   with lConditions do for ww := 0 to Count-1 do
                      if PropositionPtr(Items^[ww])^.nSentence^.FrmSort = ikFrmConj then
                             WrongSkeleton('Too complex assumption',lPos);
               {$ENDIF}
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_Propositions(lConditions);
               AReport.Out_XElEnd(elAssume);
               {$ENDIF}
               InFile.InWord;
               lFrm := ConjugatePropositions(lConditions);
               lDefs := ChopAssum(lFrm,lPos);
               {| odwrocona kolejnosc ze wzgledu na obliczanie "thesis" |}
               lConditions.Done;
```

```
      WriteThesis;
   end;
ikItmExAssumption:
   begin
      lVarBase := g.VarNbr;
      InFile.InPos(lPos);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elGiven);
      AReport.Out_XIntAttr(atNr, g.VarNbr+1);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      GetQualifiedList;
      ReadPropositions(lConditions);
      InFile.InWord;
      lFrm := xFormula(ConjugatePropositions(lConditions));
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_Propos(0, 0, CurPos, lFrm);
      {$ENDIF}
      lDefs := ChopAssum(lFrm,lPos);
      WriteQualified;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elGiven);
      {$ENDIF}
      lConditions.Done;
      for i := lVarBase+1 to g.VarNbr do FixedVar[i].nSkelConstNr := 0;
      WriteThesis;
   end;
ikItmExemplifWithEq:
   begin
      InFile.InPos(lPos);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elTakeAsVar);
      AReport.Out_XIntAttr(atNr, g.VarNbr+1);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      InFile.InWord; //'I'
      lId := InFile.Current.Nr;
      lTrm := ReadTerm;
      gFixedBase := g.VarNbr;
      inc(g.VarNbr); mizassert(2521,g.VarNbr<=MaxVarNbr);
      FixedVar[g.VarNbr].nExp := false;
      FixedVar[g.VarNbr].nIdent := lId;
      FixedVar[g.VarNbr].nTyp := GetTrmType(lTrm);
      if g.Thesis^.FrmSort = ikFrmNeg then
         if NegFrmPtr(g.Thesis)^.NegArg^.FrmSort = ikFrmFlexConj then
            g.Thesis := NewNeg(FlexFrmPtr(NegFrmPtr(g.Thesis)^.NegArg)^.nExpansion);
      g.Thesis := NewNegDis(g.Thesis);
      lDefs := ChopVars(true,true,lPos);
      g.Thesis := NewNegDis(g.Thesis);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_TypeWithId(FixedVar[g.VarNbr].nTyp,
                             FixedVar[g.VarNbr].nIdent);
      AReport.Out_Term(lTrm);
      AReport.Out_XElEnd(elTakeAsVar);
      {$ENDIF}
      WriteThesis;
      DisposeTrm(lTrm); InFile.InWord;
   end;
ikItmSimpleExemplif:
```

```
      begin
         InFile.InPos(lPos);
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart0(elTake);
         {$ENDIF}
         lTrm := ReadTerm;
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_Term(lTrm);
         AReport.Out_XElEnd(elTake);
         {$ENDIF}
         gFixedBase := g.VarNbr; inc(g.VarNbr);
         lTyp := GetTrmType(lTrm);
         FixedVar[g.VarNbr].nIdent := 0;
         FixedVar[g.VarNbr].nTyp := lTyp;
         FixedVar[g.VarNbr].nExp := false;
         if g.Thesis^.FrmSort = ikFrmNeg then
            if NegFrmPtr(g.Thesis)^.NegArg^.FrmSort = ikFrmFlexConj then
               g.Thesis := NewNeg(FlexFrmPtr(NegFrmPtr(g.Thesis)^.NegArg)^.nExpansion);
         g.Thesis := NewNegDis(g.Thesis);
         lDefs := ChopVars(true,true,lPos);
         g.Thesis := NewNegDis(g.Thesis);
         WithInFormula(g.Thesis,SetTaken);
         lTrmList := NewTrmList(lTrm,nil);
         lFrm := g.Thesis;
         if lTrmList<>InCorrTrmList then
         begin
            g.Thesis := InstFrm(g.Thesis,lTrmList);
            DisposeTrmList(lTrmList);
         end
         else g.Thesis := NewInCorFrm;
         dispose(lFrm,Done);
         dispose(FixedVar[g.VarNbr].nTyp,Done);
         dec(g.VarNbr);
         InFile.InWord;
         WriteThesis;
      end;
   ikItmConclusion:
      begin
         {$IFDEF FRM2THESIS}
             inConclusion := true;
         {$ENDIF}
             InFile.InPos(lPos);
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart0(elConclusion);
         {$ENDIF}
         InFile.InWord;
         if InFile.Current.Kind = ikBlcHereby then
            HereBy(lFrm)
         else RegularStatement(lFrm);
         lDefs := ChopConcl(lFrm,lPos);
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElEnd(elConclusion);
         {$ENDIF}
         WriteThesis;
         {$IFDEF FRM2THESIS}
         inConclusion := false;
         {$ENDIF}
      end;
   ikBlcPerCases: exit;
   ikBlcCase,ikBlcSuppose: exit;
```

```
            else Statement;
         end;
         DisplayLine(CurPos.Line,ErrorNbr);
      end;
   end;
```

This code is used in chunk 25.
Defines:
Reasoning, used in chunks 82 and 85.
Uses ChopAssum 78a, ChopConcl 77b, ChopVars 75c, ConjugatePropositions 9b, GetQualifiedList 13b, gFixedBase 12c, HereBy 96a, ReadPropositions 9a, ReadTerm 8a, RegularStatement 97, SetTaken 75b, Statement 100, WriteQualified 14b, and xFormula 13a.

82 ⟨*Per cases reasoning* 82⟩≡

```
// this is used when thesis is known
procedure PerCasesReasoning;
var
   C: LevelRec;
   itisCase: boolean;
   PerCasesPos,CasePos: Position;
   lInference: InferenceObj;
   lConditions: MCollection;
   lFrm,lThesis,llThesis,PerCasesFrm,FirstDisjunct,Thesis_Tail: FrmPtr;
   lDefNr: integer;
   lPerCasesDefs, lDefs1: NatFuncPtr; // collect Definientia numbers
label 1;
begin
   InFile.InPos(CurPos);
   PerCasesPos := CurPos;
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart(elPerCasesReasoning);
   AReport.Out_PosAsAttrs(CurPos);
   AReport.Out_XAttrEnd;
   AReport.Out_XElStart0(elBlockThesis);
   AReport.Out_Formula(g.Thesis);
   AReport.Out_XElEnd(elBlockThesis);
   {$ENDIF}
   InFile.InWord;
   LoadInferenceObj(lInference);
   InFile.InWord;
   PerCasesFrm := NewNeg(NewVerum);
   C := g; lThesis := g.Thesis^.CopyFormula;
   C.LocPredNbr := LocPredDef.Count;
   C.LocFuncNbr := LocFuncDef.Count;
   lPerCasesDefs         := new(NatFuncPtr, InitNatFunc(4,4));
   while InFile.Current.Kind <>ikMscEndBlock do
   begin
      case InFile.Current.Kind of
         ikBlcCase:
            begin
               itisCase := true;
               InFile.InPos(CurPos); CasePos := CurPos;
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XElStart(elCaseBlock);
               AReport.Out_PosAsAttrs(CurPos);
               AReport.Out_XAttrEnd;
               {$ENDIF}
               {----}
               MarkTermsInTTColl;
               {----}
               InFile.InWord;
               ReadPropositions(lConditions);
```

```
InFile.InWord;
lFrm := ConjugatePropositions(lConditions);
// the PerCasesFrm is disjunction of all cases' suppositions,
// later we must prove that such disjunction is true.
PerCasesFrm := NewDisj(PerCasesFrm,lFrm^.CopyFormula);
if lFrm^.FrmSort=ikError then lThesis := NewInCorFrm;
if lThesis^.FrmSort=ikError then
begin g.Thesis := NewInCorFrm; FirstDisjunct := NewInCorFrm; goto 1 end;
// thesis is disjunction, here we get the first disjunct,
// which is the thesis of this case; it can however get smaller
// below, by definitional expansion
Decompose(NewNegDis(lThesis),FirstDisjunct,lThesis);
FirstDisjunct := NewNegDis(FirstDisjunct);
lThesis := NewNegDis(lThesis);
g.Thesis := FirstDisjunct^.CopyFormula;
{ Tutaj jest wyjatek, bo lFrm jest kopiowana, w innjych zawolaniach
  Chopped tak nie jest.
  Problem polega na tym, ze w tym wyjatkowym wypadku,
  niemoliowsc odciecia nie powoduje bledu i dysponowanie musi byc
  dokladne !
}
{ There is an exception here because lFrm is copied,
  this is not the case in other Chopped calls.
  The problem is that in this exceptional case,
  the impossibility of the cutoff does not cause an error
  and the handling must be exact!
 }
// now try to spread (apply definiens to) and decompose
// the FirstDisjunct until lFrm (the case) can be chopped
// ##TODO: spreading is done in Chopped too, why twice?
while not Chopped(lFrm^.CopyFormula, true, lDefs1) do
begin
   dispose(lDefs1, Done); lDefs1 := nil;
   dispose(g.Thesis,Done);
   { Chopped nie dysponuje g.Thesis }
   lDefNr := SpreadAtomicFormula(FirstDisjunct,true);
   if lDefNr >= 0 then lPerCasesDefs^.Up(lDefNr);
   if FirstDisjunct^.FrmSort = ikError then
   begin
      Error(CasePos,53);
      g.Thesis := NewInCorFrm;
      lThesis := NewInCorFrm;
      dispose(lPerCasesDefs, Done);
      lPerCasesDefs := nil;
      break;
   end;
   Decompose(NewNegDis(FirstDisjunct),FirstDisjunct,Thesis_tail);
   FirstDisjunct := NewNegDis(FirstDisjunct);
   Thesis_tail := NewNegDis(Thesis_tail);
   lThesis := NewDisj(Thesis_tail,lThesis);
   g.Thesis := FirstDisjunct^.CopyFormula;
end;
// lPerCasesDefs <> nil means success above
if Assigned(lPerCasesDefs) then lPerCasesDefs^.Add(lDefs1^);
if Assigned(lDefs1) then dispose(lDefs1, Done); lDefs1 := nil;
1:
   lFrm := NewImpl(lFrm, g.Thesis^.CopyFormula);
{$IFDEF ANALYZER_REPORT}
// FirstDisjunct is conjunction of  lConditions and g.Thesis, while
// this block actually proves that lConditions imply g.Thesis -
```

```
      // that's why we have to create the thesis above (not used in preparator)
      // ##NOTE: we have three possibilities for dealing with def expansions here:
      //   (1) print them at the case block thesis
      //   (2) print them at the case item (the first item in the case block)
      //   (3) create additional 'case conclusion' skeleton item preceding the
      //       case block, and print it there
      //   (4) print them at the PerCasesJustification
      // we use (4)
      AReport.Out_XElStart0(elBlockThesis);
      AReport.Out_Formula(lFrm);
      AReport.Out_XElEnd(elBlockThesis);
      AReport.Out_XElStart0(elCase);
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elCase);
      AReport.Out_XElStart0(elThesis);
      AReport.Out_Formula(g.Thesis); // thesis after the case
      AReport.Out_NatFunc(elThesisExpansions, EmptyNatFunc);
      AReport.Out_XElEnd(elThesis);
      {$ENDIF}
      dispose(lFrm,Done);
      dispose(FirstDisjunct,Done);
      lConditions.Done;
   end;
ikBlcSuppose:
   begin
      itisCase := false;
      InFile.InPos(CurPos); CasePos := CurPos;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elSupposeBlock);
      AReport.Out_PosAsAttrs(CurPos);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      lThesis := C.Thesis^.CopyFormula;
      {----}
      MarkTermsInTTColl;
      {----}
      InFile.InWord;
      ReadPropositions(lConditions);
      InFile.InWord;
      lFrm := ConjugatePropositions(lConditions);
      // llThesis is the 'real' thesis of this suppose block
      llThesis := NewImpl(lFrm^.CopyFormula, lThesis^.CopyFormula);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elBlockThesis);
      AReport.Out_Formula(llThesis);
      AReport.Out_XElEnd(elBlockThesis);
      AReport.Out_XElStart0(elSuppose);
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elSuppose);
      AReport.Out_XElStart0(elThesis);
      AReport.Out_Formula(lThesis); // thesis after the suppose
      AReport.Out_NatFunc(elThesisExpansions, EmptyNatFunc);
      AReport.Out_XElEnd(elThesis);
      {$ENDIF}
      dispose(llThesis, Done);
      PerCasesFrm := NewDisj(PerCasesFrm,lFrm);
      g.Thesis := lThesis;
      lConditions.Done;
      lThesis := lThesis^.CopyFormula;
   end;
```

```
      else RuntimeError(2641);
      end;
      DisplayLine(CurPos.Line,ErrorNbr);
      Reasoning;
      if InFile.Current.Kind = ikBlcPerCases
      then PerCasesReasoning
      else if (g.Thesis^.FrmSort<>ikFrmVerum) and (g.Thesis^.FrmSort<>ikError)
      then Error(CasePos,60);
      mizassert(2310,InFile.Current.Kind = ikMscEndBlock);
      InFile.InPos(CurPos); InFile.InWord;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_EndPos(CurPos);
      if itisCase then AReport.Out_XElEnd(elCaseBlock)
      else AReport.Out_XElEnd(elSupposeBlock);
      {$ENDIF}
      dispose(g.Thesis,Done);
      DisposeLevel(c);
   end;
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart0(elPerCases);
   AReport.Out_Propos(0, 0, CurPos, PerCasesFrm);
   AReport.Out_Inference(lInference);
   AReport.Out_XElEnd(elPerCases);
   AReport.Out_XElStart0(elThesis);
   // thesis after the per cases, this is broken now:
   // for Case blocks, it should be the conjunction
   // of cases theses as implications (rather than cases' theses as
   // conjuctions); for Suppose blocks, it should also rather be
   // conjunction of implications (cases' theses)
   AReport.Out_Formula(lThesis);
   if not Assigned(lPerCasesDefs) then
      lPerCasesDefs      := new(NatFuncPtr, InitNatFunc(0,0));
   // definientia used for cases
   AReport.Out_NatFunc(elThesisExpansions, lPerCasesDefs^);
   AReport.Out_XElEnd(elThesis);
   AReport.Out_EndPos(CurPos);
   AReport.Out_XElEnd(elPerCasesReasoning);
   {$ENDIF}
   dispose(lPerCasesDefs, Done);
   dispose(PerCasesFrm,Done);
   lInference.Done;
   if itisCase then
      if lThesis^.FrmSort <> ikError then
         if lThesis^.FrmSort = ikFrmNeg then
         begin if NegFrmPtr(lThesis)^.NegArg^.FrmSort<> ikFrmVerum then ErrImm(54) end
         else  ErrImm(54);
   dispose(lThesis,Done);
end;
```

This code is used in chunk 25.
Defines:
PerCasesReasoning, used in chunk 85.
Uses Chopped 73, ConjugatePropositions 9b, Decompose 72a, DisposeLevel 78b, ReadPropositions 9a, Reasoning 78c, and SpreadAtomicFormula 72b.

85 ⟨*Demonstration* 85⟩≡

```
procedure Demonstration(ThesisId,fLabId: integer; fThesis: FrmPtr);
var
   L: LevelRec;
   Thesis_head: FrmPtr;
   ii: integer;
label Finished;
```

```
begin
   {$IFDEF ANALYZER_REPORT}
   // making things compatible with simplejustification;
   // the position is probably unnecessary
   AReport.Out_Propos(ThesisId, fLabId, CurPos, fThesis);
   AReport.Out_XElStart(elProof);
   if ThesisId <> 0 then
   begin
      AReport.Out_XIntAttr(atNr, ThesisId);
      AReport.Out_XIntAttr(atVid, fLabId);
   end;
   AReport.Out_PosAsAttrs(CurPos);
   AReport.Out_XAttrEnd;
   AReport.Out_XElStart0(elBlockThesis);
   AReport.Out_Formula(fThesis);
   AReport.Out_XElEnd(elBlockThesis);
   {$ENDIF}
   L := g;
   L.LocPredNbr := LocPredDef.Count;
   L.LocFuncNbr := LocFuncDef.Count;
   {----}
   MarkTermsInTTColl;
   {----}
   g.Thesis := fThesis^.CopyFormula;
   Reasoning;
   if InFile.Current.Kind = ikBlcPerCases then
   begin
      PerCasesReasoning;
      InFile.InPos(CurPos); InFile.InWord; // ikMscEndBlock
   end
   else
   begin
      InFile.InPos(CurPos); InFile.InWord; // ikMscEndBlock
      if (g.Thesis^.FrmSort<>ikFrmVerum) and (g.Thesis^.FrmSort<>ikError) then
      begin
         Decompose(g.Thesis,Thesis_head,g.Thesis);
         for ii := 1 to MaxExpansionNbr do
         begin
            if Thesis_head^.FrmSort = ikFrmVerum then goto Finished;
            // ##TODO: this is never used in MML 853, works only when
            //  someone defined a predicate as 'not contradiction', and
            // the test for only the head being true is very fragile and risky.
            // It should be removed, and def expansions are not collected
            // from it, since I do not want to introduce additional
            // overhead for keeping them at block thesis just because of
            // such rubbish.
            SpreadAtomicFormula(Thesis_head,true);
            if Thesis_head^.FrmSort = ikError then break;
            Decompose(NewConj(Thesis_head,g.Thesis),Thesis_head,g.Thesis);
         end;
         if not AxiomsAllowed then
            ErrImm(70);
      end;
      Finished:
         dispose(g.Thesis,Done);
   end;
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_EndPos(CurPos);
   AReport.Out_XElEnd(elProof);
   {$ENDIF}
```

```
      DisposeLevel(L);
   end;
```

This code is used in chunk 25.
Defines:
Demonstration, used in chunks 96b, 97, and 134.
Uses Decompose 72a, DisposeLevel 78b, MaxExpansionNbr 73, PerCasesReasoning 82, Reasoning 78c, and SpreadAtomicFormula 72b.

87a ⟨*Change declared constant to bound variable* 87a⟩≡

```
procedure ChangeDeclConstToBound(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
   with VarTrmPtr(fTrm)^ do
      case TrmSort of
         ikTrmLocus: TrmSort := ikTrmBound;
         ikTrmBound: inc(VarNr,g.GenCount);
         ikTrmConstant:
            if (VarNr>g.DemBase) and (FixedVar[VarNr].nSkelConstNr<>0) then
            begin VarNr := FixedVar[VarNr].nSkelConstNr; TrmSort := ikTrmBound end;
         ikTrmIt:
            begin lTrm := fTrm; fTrm := NewVarTrm(ikTrmBound,g.GenCount);
            dispose(lTrm,Done);
            end;
      end;
end;
```

This code is used in chunk 25.
Defines:
ChangeDeclConstToBound, used in chunks 87b, 88a, and 115–17.

RSNENTRY is defined in generato.pas, which also defines SkList as an MCollection of the types of local constants.

87b ⟨*Skeletonize list* 87b⟩≡

```
procedure SkelList(FF: char; frst: integer);
var
   lEntry: RSNENTRY;
   k: integer;
   lTyp: TypPtr;
begin
   new(lEntry);
   with lEntry^ do
   begin
      PreviousEntry := g.LastEntry;
      FORM := FF;
      SkList.Init(g.VarNbr-FRST,0);
      SkOrigTyps.Init(g.VarNbr-FRST,0);
      SkIdents.Init(g.VarNbr-FRST);
      SkFrstConstNr := frst;
      for K := FRST+1 to g.VarNbr do
      begin
         lTyp := FixedVar[k].nTyp^.CopyType;
         gConstErr := false;
         lTyp^.WithInType(CheckLocConst);
         if gConstErr then
         begin ErrImm(50); dispose(lTyp,Done); lTyp := NewIncorTyp end;
         if lTyp^.TypSort=ikError then begin g.Err := true; exit end;
         SkOrigTyps.Insert(lTyp^.CopyType);
         lTyp^.WithInType(ChangeDeclConstToBound);
         SkList.Insert(lTyp);
         SkIdents.Insert(FixedVar[k].nIdent);
      end;
```

```
      end;
      g.LastEntry := lEntry;
   end;
```

This code is used in chunk 25.
Defines:
SkelList, used in chunks 88b and 121.
Uses ChangeDeclConstToBound 87a and gConstErr 17a.

88a ⟨*Skeletonize sentence* 88a⟩≡

```
   procedure SkelSnt(FF: char; fFrm: FrmPtr);
   var
      lEntry: RSNENTRY;
   begin
      gConstErr := false;
      WithInFormula(fFrm,CheckLocConst);
      if gConstErr then begin ErrImm(68); g.Err := true; exit end;
      if fFrm^.FrmSort=ikError then begin g.Err := true; exit end;
      new(lEntry);
      with lEntry^ do
      begin
         PreviousEntry := g.LastEntry;
         FORM := FF;
         SkSnt := fFrm;
         DSnt := fFrm^.CopyFormula;
         WithInFormula(fFrm,ChangeDeclConstToBound);
      end;
      g.LastEntry := lEntry;
   end;
```

This code is used in chunk 25.
Defines:
SkelSnt, used in chunks 88b, 92, and 121.
Uses ChangeDeclConstToBound 87a and gConstErr 17a.

88b ⟨*Diffuse Reasoning* 88b⟩≡

```
   procedure DiffReasoning;
   var
      lVarBase, i, lId: integer;
      lTrm: TrmPtr;
      lPos: Position;
      lConditions: MCollection;
      lFrm: FrmPtr;
   begin
      while InFile.Current.Kind <> ikMscEndBlock do
      begin
         case InFile.Current.Kind of
            ikItmGeneralization:
               begin
                  InFile.InPos(CurPos);
                  {$IFDEF ANALYZER_REPORT}
                  AReport.Out_XElStart(elLet);
                  AReport.Out_XIntAttr(atNr, g.VarNbr+1);
                  AReport.Out_XAttrEnd;
                  {$ENDIF}
                  GetQualifiedList;
                  for i := gFixedBase+1 to g.VarNbr do
                  begin
                     inc(g.GenCount);
                     FixedVar[i].nSkelConstNr := g.GenCount;
                  end;
                  WriteQualified;
                  {$IFDEF ANALYZER_REPORT}
```

```
      AReport.Out_XElEnd(elLet);
      {$ENDIF}
      SkelList('D',gFixedBase);
   end;
ikItmAssumption:
   begin
      InFile.InPos(lPos);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elAssume);
      {$ENDIF}
      InFile.InWord; ReadPropositions(lConditions);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elAssume);
      {$ENDIF}
      InFile.InWord;
      lFrm := ConjugatePropositions(lConditions);
      SkelSnt('A',lFrm);
      lConditions.Done;
   end;
ikItmExAssumption:
   begin
      lVarBase := g.VarNbr;
      InFile.InPos(CurPos);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elGiven);
      AReport.Out_XIntAttr(atNr, g.VarNbr+1);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      GetQualifiedList;
      ReadPropositions(lConditions);
      InFile.InWord;
      lFrm := xFormula(ConjugatePropositions(lConditions));
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_Propos(0, 0, CurPos, lFrm);
      {$ENDIF}
      SkelSnt('A',lFrm);
      WriteQualified;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elGiven);
      {$ENDIF}
      lConditions.Done;
      for i := lVarBase+1 to g.VarNbr do FixedVar[i].nSkelConstNr := 0;
   end;
ikItmExemplifWithEq:
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elTakeAsVar);
      AReport.Out_XIntAttr(atNr, g.VarNbr+1);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      InFile.InPos(lPos);
      InFile.InWord; //'I'
      lId := InFile.Current.Nr;
      lTrm := ReadTerm;
      inc(g.VarNbr); mizassert(2521,g.VarNbr<=MaxVarNbr);
      FixedVar[g.VarNbr].nExp := false;
      FixedVar[g.VarNbr].nIdent := lId;
      FixedVar[g.VarNbr].nTyp := GetTrmType(lTrm);
```

```
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_TypeWithId(FixedVar[g.VarNbr].nTyp,
                                       FixedVar[g.VarNbr].nIdent);
                AReport.Out_Term(lTrm);
                AReport.Out_XElEnd(elTakeAsVar);
                {$ENDIF}
                inc(g.GenCount);
                FixedVar[g.VarNbr].nSkelConstNr := g.GenCount;
                SkelList('C',g.VarNbr-1);
                DisposeTrm(lTrm); InFile.InWord;
             end;
          ikItmSimpleExemplif:  // probably forbidden without equality
             begin
                InFile.InPos(CurPos);
                ErrImm(64);
                g.Err := true;
                lTrm := ReadTerm;
                DisposeTrm(lTrm); InFile.InWord;
             end;
          ikItmConclusion:
             begin
                {$IFDEF FRM2THESIS}
                inConclusion := true;
                {$ENDIF}
                InFile.InPos(CurPos);
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElStart0(elConclusion);
                {$ENDIF}
                InFile.InWord;
                if InFile.Current.Kind = ikBlcHereby then
                   HereBy(lFrm)
                else RegularStatement(lFrm);
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElEnd(elConclusion);
                {$ENDIF}
                SkelSnt('B',lFrm);
                {$IFDEF FRM2THESIS}
                inConclusion := false;
                {$ENDIF}
             end;
          ikBlcPerCases,ikBlcCase,ikBlcSuppose: exit;
        else Statement;
        end;
        DisplayLine(CurPos.Line,ErrorNbr);
     end;
  end;
```

This code is used in chunk 25.

Defines:

DiffReasoning, used in chunks 92 and 95.

Uses ConjugatePropositions 9b, GetQualifiedList 13b, gFixedBase 12c, HereBy 96a, ReadPropositions 9a, ReadTerm 8a, RegularStatement 97, SkelList 87b, SkelSnt 88a, Statement 100, WriteQualified 14b, and xFormula 13a.

90 ⟨*New list of universally quantified variables* 90⟩≡

```
  function NewUnivList(const FL: MCollection;
                       const Ids: IntSequence;
                       fFrm: FrmPtr): FrmPtr;
  var
     k: integer;
  begin
     if fFrm^.FrmSort=ikError then begin NewUnivList := NewInCorFrm; exit end;
     with FL do
```

```
      for k := Count-1 downto 0 do
         fFrm := NewUnivI(Ids.Value(k),TypPtr(Items^[k])^.CopyType,fFrm);
   NewUnivList := fFrm;
end;
```

This code is used in chunk 25.
Defines:
NewUnivList, used in chunks 91c and 119.

91a ⟨*Change skeletonized fixed variable to bound variable* 91a⟩≡

```
var gSkListCount, gSkFrstConstNr: integer;
procedure ChangeSkFixedToBound(var fTrm: TrmPtr);
begin
   with VarTrmPtr(fTrm)^ do
      case TrmSort of
         ikTrmBound: inc(VarNr, gSkListCount);
         ikTrmConstant:
            if VarNr> gSkFrstConstNr then
            begin TrmSort := ikTrmBound; dec(VarNr,gSkFrstConstNr) end;
      end;
end;
```

This code is used in chunk 25.
Defines:
ChangeSkFixedToBound, used in chunk 91b.
gSkFrstConstNr, used in chunk 91b.
gSkListCount, used in chunk 91b.

91b ⟨*New universal list (one)* 91b⟩≡

```
// version gradually fixing local consts, needed for proper from of subtheses
function NewUnivList1(const FL: MCollection;
                      const Ids: IntSequence;
                      fFrm: FrmPtr;
                      var fFrstConstNr: integer): FrmPtr;
var
   k: integer;
   lTyp: TypPtr;
begin
   if fFrm^.FrmSort=ikError then begin NewUnivList1 := NewInCorFrm; exit end;
   gSkFrstConstNr := fFrstConstNr;
   gSkListCount := FL.Count;
   WithInFormula(fFrm, ChangeSkFixedToBound);
   with FL do
      for k := Count-1 downto 0 do
      begin
         dec(gSkListCount); // needed for ChangeSkFixedToBound in the type
         lTyp := TypPtr(Items^[k])^.CopyType;
         lTyp^.WithinType(ChangeSkFixedToBound);
         fFrm := NewUnivI(Ids.Value(k), lTyp, fFrm);
      end;
   NewUnivList1 := fFrm;
end;
```

This code is used in chunk 25.
Defines:
NewUnivList1, used in chunk 91c.
Uses ChangeSkFixedToBound 91a, gSkFrstConstNr 91a, and gSkListCount 91a.

91c ⟨*Reasoning result* 91c⟩≡

```
function ReasResult(fFrm: FrmPtr; var fSubResults: MCollection): FrmPtr;
var
   lEntry: RSNENTRY;
   lFrm: FrmPtr;
begin
```

```
   fSubResults.Init(4,4);
   lFrm := fFrm^.CopyFormula;
   if g.Err then begin ReasResult := NewInCorFrm; exit end;
   while g.LastEntry <> nil do
      with g.LastEntry^ do
   begin
      fSubResults.Insert(lFrm^.CopyFormula);
      case FORM of
         'A': begin fFrm := NewImpl(SkSnt,fFrm); lFrm := NewImpl(dSnt,lFrm); end;
         'B': begin fFrm := NewConj(SkSnt,fFrm); lFrm := NewConj(dSnt,lFrm); end;
         'C':
            begin
               fFrm := NewNeg(NewUnivList(SkList,SkIdents,NewNegDis(fFrm)));
               lFrm := NewNeg(NewUnivList1(SkOrigTyps,SkIdents,NewNegDis(lFrm),SkFrstConstNr));
               SkList.Done; SkIdents.Done; SkOrigTyps.Done;
            end;
         'D':
            begin
               fFrm := NewUnivList(SkList,SkIdents, fFrm);
               lFrm := NewUnivList1(SkOrigTyps,SkIdents, lFrm, SkFrstConstNr);
               SkList.Done; SkIdents.Done; SkOrigTyps.Done;
            end;
      else RunTimeError(2008);
      end;
      lEntry := PreviousEntry;
      dispose(g.LastEntry);
      g.LastEntry := lEntry;
   end;
   dispose(lFrm,Done);
   ReasResult := fFrm;
end;
```

This code is used in chunk 25.
Defines:
ReasResult, used in chunks 92 and 95.
Uses NewUnivList 90 and NewUnivList1 91b.

92 ⟨*Diffuse per cases reasoning* 92⟩≡

```
procedure DiffPerCasesReasoning(var fResult: FrmPtr);
var
   C: LevelRec;
   lConditions,lSubResults: MCollection;
   lFrm,lGuard,lPerCases,lResult,lPerCasesResult,llThesis,llGuard: FrmPtr;
   lInference: InferenceObj;
   z: integer;
begin
   InFile.InPos(CurPos);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart(elPerCasesReasoning);
   AReport.Out_PosAsAttrs(CurPos);
   AReport.Out_XAttrEnd;
   {$ENDIF}
   InFile.InWord;
   LoadInferenceObj(lInference);
   InFile.InWord;
   lPerCases := NewNeg(NewVerum);
   C := g;
   C.LocPredNbr := LocPredDef.Count;
   C.LocFuncNbr := LocFuncDef.Count;
   case InFile.Current.Kind of
      ikBlcCase:
         begin lResult := NewNeg(NewVerum);
```

```
      {----}
      MarkTermsInTTColl;
      {----}
      repeat InFile.InPos(CurPos);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elCaseBlock);
      AReport.Out_PosAsAttrs(CurPos);
      AReport.Out_XAttrEnd;
      {$ENDIF}
      InFile.InWord;
      ReadPropositions(lConditions);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elCase);
      AReport.Out_Propositions(lConditions);
      AReport.Out_XElEnd(elCase);
      {$ENDIF}
      InFile.InWord;
      lGuard := ConjugatePropositions(lConditions);
      lPerCases := NewDisj(lPerCases,lGuard^.CopyFormula);
      g.Err := false; g.LastEntry := nil;
      SkelSnt('B',lGuard^.CopyFormula);
      DiffReasoning;
      lFrm := ReasResult(NewVerum, lSubResults);
      if InFile.Current.Kind = ikBlcPerCases then
      begin
         DiffPerCasesReasoning(lPerCasesResult);
         if lPerCasesResult^.FrmSort=ikError then C.Err := true
         else lFrm := NewConj(lFrm,lPerCasesResult);
      end;
      InFile.InPos(CurPos); InFile.InWord;
      // llThesis is the 'real' thesis of this case block, i.e.
      // an implication, not conjunction
      llThesis := NewImpl(lGuard, lFrm^.CopyFormula);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_EndPos(CurPos);
      AReport.Out_XElStart0(elBlockThesis);
      for z := lSubResults.Count-1 downto 0 do
      begin
         AReport.Out_XElStart0(elThesis);
         AReport.Out_Formula(lSubResults.Items^[z]);
         AReport.Out_NatFunc(elThesisExpansions, EmptyNatFunc);
         AReport.Out_XElEnd(elThesis);
      end;
      AReport.Out_Formula(llThesis);
      // just as Thesis now, stdprep has to produce a proposition
      AReport.Out_XElEnd(elBlockThesis);
      AReport.Out_XElEnd(elCaseBlock);
      {$ENDIF}
      lSubResults.Done;
      lConditions.Done;
      dispose(llThesis, Done);
      if lFrm^.FrmSort=ikError then C.Err := true
      else lResult := NewDisj(lResult,lFrm);
      DisposeLevel(c);
      until InFile.Current.Kind <> ikBlcCase;
      end;
   ikBlcSuppose:
      begin
         lResult := nil;
         {----}
```

```
MarkTermsInTTColl;
{----}
repeat InFile.InPos(CurPos);
{$IFDEF ANALYZER_REPORT}
AReport.Out_XElStart(elSupposeBlock);
AReport.Out_PosAsAttrs(CurPos);
AReport.Out_XAttrEnd;
{$ENDIF}
InFile.InWord;
ReadPropositions(lConditions);
{$IFDEF ANALYZER_REPORT}
AReport.Out_XElStart0(elSuppose);
AReport.Out_Propositions(lConditions);
AReport.Out_XElEnd(elSuppose);
{$ENDIF}
InFile.InWord;
lGuard := ConjugatePropositions(lConditions);
lPerCases := NewDisj(lPerCases,lGuard^.CopyFormula);
g.Err := false; g.LastEntry := nil;
DiffReasoning;
lFrm := ReasResult(NewVerum,lSubResults);
if InFile.Current.Kind = ikBlcPerCases then
begin
   DiffPerCasesReasoning(lPerCasesResult);
   if lPerCasesResult^.FrmSort=ikError then C.Err := true
   else lFrm := NewConj(lFrm,lPerCasesResult);
end;
InFile.InPos(CurPos); InFile.InWord;
// llThesis is the 'real' thesis of this suppose block
llThesis := NewImpl(lGuard, lFrm^.CopyFormula);
{$IFDEF ANALYZER_REPORT}
AReport.Out_EndPos(CurPos);
AReport.Out_XElStart0(elBlockThesis);
// lFrm is the thesis after suppose, hence it's added
// as the last subresult
lSubResults.Insert(lFrm);
for z := lSubResults.Count-1 downto 0 do
begin
   AReport.Out_XElStart0(elThesis);
   AReport.Out_Formula(lSubResults.Items^[z]);
   AReport.Out_NatFunc(elThesisExpansions, EmptyNatFunc);
   AReport.Out_XElEnd(elThesis);
end;
AReport.Out_Formula(llThesis);
// just as Thesis now, stdprep has to produce a proposition
AReport.Out_XElEnd(elBlockThesis);
AReport.Out_XElEnd(elSupposeBlock);
{$ENDIF}
dec(lSubResults.Count); // not to dispose lFrm
lSubResults.Done;
lConditions.Done;
dispose(llThesis, Done);
if lFrm^.FrmSort=ikError then C.Err := true;
//     else
if lResult = nil then lResult := lFrm
else
begin
   if not EqFrm(lResult,lFrm) then ErrImm(59);
   dispose(lFrm,Done);
end;
```

```
            DisposeLevel(c);
            until InFile.Current.Kind <> ikBlcSuppose;
         end;
    else RunTimeError(2493);
    end;
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_XElStart0(elPerCases);
    AReport.Out_Propos(0, 0, CurPos, lPerCases);
    AReport.Out_Inference(lInference);
    AReport.Out_XElEnd(elPerCases);
    AReport.Out_EndPos(CurPos);
    AReport.Out_XElStart0(elBlockThesis);
    // just as Thesis now, stdprep has to produce a proposition
    // ###TODO: BUG: lResult can be nil here in incorrect percases,
    //          which prevents Out_Formula; fix that
    if Assigned(lResult) then AReport.Out_Formula(lResult)
    else AReport.Out_Formula(NewInCorFrm);
    AReport.Out_XElEnd(elBlockThesis);
    AReport.Out_XElEnd(elPerCasesReasoning);
    {$ENDIF}
    dispose(lPerCases,Done);
    lInference.Done;
    fResult := lResult;
  end;
```

This code is used in chunk 25.
Defines:
DiffPerCasesReasoning, used in chunk 95.
Uses ConjugatePropositions 9b, DiffReasoning 88b, DisposeLevel 78b, ReadPropositions 9a, ReasResult 91c, and SkelSnt 88a.

95 ⟨*Diffuse statement* 95⟩≡

```
  procedure DiffuseStatement(var fResult: FrmPtr; var fSubResults: MCollection);
  var
     L: LevelRec;
     lResult: FrmPtr;
  begin
     L := g;
     L.LocPredNbr := LocPredDef.Count;
     L.LocFuncNbr := LocFuncDef.Count;
     {----}
     MarkTermsInTTColl;
     {----}
     g.LastEntry := nil;
     g.Err := false;
     g.GenCount := 0;
     g.DemBase := g.VarNbr;
     Infile.InWord;
     DiffReasoning;
     if InFile.Current.Kind = ikBlcPerCases then
        DiffPerCasesReasoning(lResult)
     else lResult := NewVerum;
     fResult := ReasResult(lResult, fSubResults);
     DisposeLevel(L);
     InFile.InPos(CurPos);
  end;
```

This code is used in chunk 25.
Defines:
DiffuseStatement, used in chunks 96a and 97.
Uses DiffPerCasesReasoning 92, DiffReasoning 88b, DisposeLevel 78b, and ReasResult 91c.

96a ⟨*Hereby* 96a⟩≡

```
procedure HereBy(var fFrm: FrmPtr);
var
   lFrm: FrmPtr;
   lSubResults: MCollection;
   z: integer;
begin
   InFile.InPos(CurPos);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart(elNow);
   AReport.Out_PosAsAttrs(CurPos);
   AReport.Out_XAttrEnd;
   {$ENDIF}
   DiffuseStatement(lFrm, lSubResults); InFile.InWord;
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_EndPos(CurPos);
   // just as Thesis now, stdprep has to produce a proposition
   AReport.Out_XElStart0(elBlockThesis);
   for z := lSubResults.Count-1 downto 0 do
   begin
      AReport.Out_XElStart0(elThesis);
      AReport.Out_Formula(lSubResults.Items^[z]);
      AReport.Out_NatFunc(elThesisExpansions, EmptyNatFunc);
      AReport.Out_XElEnd(elThesis);
   end;
   AReport.Out_Formula(lFrm);
   AReport.Out_XElEnd(elBlockThesis);
   AReport.Out_XElEnd(elNow);
   {$ENDIF}
   lSubResults.Done;
   fFrm := lFrm;
end;
```

This code is used in chunk 25.
Defines:
HereBy, used in chunks 78c and 88b.
Uses DiffuseStatement 95.

96b ⟨*Justify* 96b⟩≡

```
var gInference: InferenceObj;

procedure Justify(ThesisId, fLabId: integer; fThesis: FrmPtr);
begin
   InFile.InWord;
   case InFile.Current.Kind of
      '"':
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_Propos(ThesisId, fLabId, CurPos, fThesis);
            {$ENDIF}
            LoadInferenceObj(gInference);
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_Inference(gInference);
            {$ENDIF}
            gInference.Done;
            InFile.InWord;
         end;
      ikBlcProof:
         begin
            InFile.InPos(CurPos);
            Infile.InWord;
            Demonstration(ThesisId,fLabId,fThesis);
```

```
            end;
         else
            begin
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_Propos(ThesisId, fLabId, CurPos, fThesis);
               AReport.Out_XEl1(elSkippedProof);
               {$ENDIF}
            end;
      end;
   end;
```

This code is used in chunk 25.
Defines:
gInference, used in chunk 97.
Justify, used in chunks 24, 100, 106b, 129, and 135b.
Uses Demonstration 85.

97 ⟨*Regular statement* 97⟩≡

```
   // RegularStatement here can be:
   // DiffuseStatement, IterativeEquality, or Statement proved
   // by Simplejustification or Proof, or @Proof
   procedure RegularStatement(var fFrm: FrmPtr);
   var
      lPred,i,z,lLabId: integer;
      lLabel: Lexem;
      lFrm: FrmPtr;
      lArgs: TrmList;
      LeftSide,lTrm: TrmPtr;
      lIterSteps,lSubResults: MCollection;
      {$IFDEF FRM2THESIS}
      StartPos, EndPos: Position;
      {$ENDIF}
   label OK;
   begin
      lLabel := InFile.Current;
      InFile.InInt(lLabId);
      InFile.InPos(CurPos);
      {$IFDEF FRM2THESIS}
      StartPos := CurPos;
      {$ENDIF}
      InFile.InWord;
      if InFile.Current.Kind=ikBlcDiffuse then
      begin
         InFile.InPos(CurPos);
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elNow);
         if lLabel.Nr <> 0 then
         begin
            AReport.Out_XIntAttr(atNr, lLabel.Nr);
            AReport.Out_XIntAttr(atVid, lLabId);
         end;
         AReport.Out_PosAsAttrs(CurPos);
         AReport.Out_XAttrEnd;
         {$ENDIF}
         DiffuseStatement(lFrm,lSubResults); InFile.InWord;
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_EndPos(CurPos);
         // just as Thesis now, stdprep has to produce a proposition;
         // the temporary theses are printed in reverse order, so that they
         // correspond to the order of skeleton items;
         AReport.Out_XElStart0(elBlockThesis);
         for z := lSubResults.Count-1 downto 0 do
```

```
      begin
         AReport.Out_XElStart0(elThesis);
         AReport.Out_Formula(lSubResults.Items^[z]);
         AReport.Out_NatFunc(elThesisExpansions, EmptyNatFunc);
         AReport.Out_XElEnd(elThesis);
      end;
      AReport.Out_Formula(lFrm);
      AReport.Out_XElEnd(elBlockThesis);
      AReport.Out_XElEnd(elNow);
      {$ENDIF}
      lSubResults.Done;
      fFrm := lFrm;
   end
   else
   begin
      lFrm := ReadSentence(false);
      InFile.InWord;
      {$IFDEF FRM2THESIS}
      EndPos := CurPos;
      {$ENDIF}
      case InFile.Current.Kind of
         '"':
            begin
               LoadInferenceObj(gInference);
               InFile.InWord;
               if InFile.Current.Kind = ikItmIterEquality then
               begin
                  with lFrm^ do
                     if FrmSort=ikFrmPred then
                     begin AdjustFrm(PredFrmPtr(lFrm),lPred,lArgs);
                     if lPred=gBuiltIn[rqEqualsTo] then
                     begin
                        LeftSide := CopyTerm(lArgs^.XTrmPtr);
                        lTrm := CopyTerm(lArgs^.NextTrm^.XTrmPtr);
                        goto OK;
                     end;
                     end;
                  if lFrm^.FrmSort<>ikError then ErrImm(159);
                  LeftSide := NewIncorTrm; lTrm := NewIncorTrm;
                  OK:
                     dispose(lFrm,Done);
                  lIterSteps.Init(4,4);
                  lIterSteps.Insert(new(IterStepPtr,Init(lTrm,gInference)));
                  repeat
                     InFile.InPos(CurPos);
                     lTrm := ReadTerm; InFile.InWord;
                     LoadInferenceObj(gInference);
                     InFile.InWord;
                     lIterSteps.Insert(new(IterStepPtr,Init(lTrm,gInference)));
                  until InFile.Current.Kind <> ikItmIterEquality;
                  {$IFDEF ANALYZER_REPORT}
                  AReport.Out_XElStart(elIterEquality);
                  if lLabel.Nr <> 0 then
                  begin
                     AReport.Out_XIntAttr(atNr, lLabel.Nr);
                     AReport.Out_XIntAttr(atVid, lLabId);
                  end;
                  AReport.Out_PosAsAttrs(CurPos);
                  AReport.Out_XAttrEnd;
                  AReport.Out_Term(LeftSide);
```

```
                  for i := 0 to lIterSteps.Count-1 do
                     AReport.Out_IterStep(IterStepPtr(lIterSteps.Items^[i])^);
                  AReport.Out_XElEnd(elIterEquality);
                  {$ENDIF}
                  fFrm := NewEqFrm(LeftSide,CopyTerm(lTrm));
                  lIterSteps.Done;
               end
               else
               begin
                  {$IFDEF ANALYZER_REPORT}
                  AReport.Out_Propos(lLabel.Nr, lLabId, CurPos, lFrm);
                  AReport.Out_Inference(gInference);
                  {$ENDIF}
                  fFrm := lFrm;

                  {$IFDEF FRM2THESIS}

                  {$IFDEF MDEBUG}
                  writeln(infofile,'START');
                  write(infofile,'g.Thesis=');
                  if g.Thesis <> nil then InfoFormula(g.Thesis);
                  writeln(infofile,' ');
                  write(infofile,'fFrm=');
                  if fFrm <> nil then InfoFormula(fFrm);
                  writeln(infofile,' ');
                  writeln(infofile,' ');
                  {$ENDIF}
                  if g.Thesis <> nil then
                     if not inSchemeInfer then
                        if inConclusion then
                        begin
                           //            if StrictEqFrm(fFrm,g.Thesis) then Error(StartPos,1000); // it's possibl
                           if g.Thesis^.FrmSort = '%' then Error(StartPos,1001); // unnecessary 'thus thesis;'
                        end;
                  {$ENDIF}

                  gInference.Done;
               end;
            end;
         ikBlcProof:
            begin
               fFrm := lFrm;
               InFile.InPos(CurPos);
               Infile.InWord;
               Demonstration(lLabel.Nr,lLabId,lFrm);
            end;
      else
         // old Preparator just accepts such statements - a bit risky
      begin
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_Propos(lLabel.Nr, lLabId, CurPos, lFrm);
         AReport.Out_XEl1(elSkippedProof);
         {$ENDIF}
         fFrm := lFrm;
      end;
      end;
   end;
end;
```

This code is used in chunk 25.
Defines:

RegularStatement, used in chunks 71, 78c, 88b, and 100.
Uses Demonstration 85, DiffuseStatement 95, gInference 96b, ReadSentence 7a, ReadTerm 8a, and Statement 100.

100 ⟨*Parse a statement* 100⟩≡

```
procedure Statement;
var
   LocBase,i,lVarBase,lId: integer;
   lFrm: FrmPtr;
   lTrm: TrmPtr;
   lConditions: MCollection;
   lTyp: TypPtr;
   lArgs: MList;
   lExpPtr: ExpPtr;
begin
   lVarBase := g.VarNbr;
   case InFile.Current.Kind of
      ikItmPrivConstant:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elSet);
            AReport.Out_XIntAttr(atNr, g.VarNbr+1);
            AReport.Out_XAttrEnd;
            {$ENDIF}
            InFile.InWord; //'I'
            lId := InFile.Current.Nr;
            lTrm := ReadTerm; InFile.InWord;
            inc(g.VarNbr); mizassert(2520,g.VarNbr<=MaxVarNbr);
            FixedVar[g.VarNbr].nIdent := lId;
            FixedVar[g.VarNbr].nExp := true;
            FixedVar[g.VarNbr].nDef := lTrm;
            FixedVar[g.VarNbr].nTyp := CopyTrmType(lTrm);
            { To jest chyba tylko do zachowania numeracji lub na wszelki wypadek }
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_Term(lTrm);
            AReport.Out_TypeWithId(FixedVar[g.VarNbr].nTyp,
                                   FixedVar[g.VarNbr].nIdent);
            AReport.Out_XElEnd(elSet);
            {$ENDIF}
         end;
      ikItmPrivFunc:
         begin
            {----}
            MarkTermsInTTColl;
            {----}
            InFile.InWord;
            lId := InFile.Current.Nr;
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefFunc);
            {$ENDIF}
            AnalizeArgTypeList(lArgs);
            lTrm := ReadTerm;
            lTyp := GetTrmType(lTrm);
            if lTyp^.TypSort=ikError then
               lTyp := AnyTyp^.CopyType;
            LocFuncDef.Insert(new(FuncDefPtr,Init(lId,lArgs,lTrm,lTyp)));
            InFile.InWord;
            with LocFuncDef,FuncDefPtr(Items^[Count-1])^ do
            begin
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XIntAttr(atNr, Count);
```

```
            AReport.Out_XIntAttr(atVid, lId);
            AReport.Out_XAttrEnd;
            AReport.Out_ArgTypes(fPrimaries);
            AReport.Out_Term(fFuncDef);
            AReport.Out_Type(fFuncTyp);
            AReport.Out_XElEnd(elDefFunc);
            {$ENDIF}
         end;
         RemoveTermsFromTTColl;
      end;
   ikItmPrivPred:
      begin
         {----}
         MarkTermsInTTColl;
         {----}
         InFile.InWord;
         lId := InFile.Current.Nr;
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elDefPred);
         {$ENDIF}
         AnalizeArgTypeList(lArgs);
         BoundVarNbr := 0;
         lFrm := ReadSentence(false);
         InFile.InWord;
         RemoveTermsFromTTColl;
         LocPredDef.Insert(new(LocPredDefPtr,Init(lId,lArgs,lFrm)));
         {$IFDEF ANALYZER_REPORT}
         with LocPredDef,LocPredDefPtr(Items^[Count-1])^ do
         begin
            AReport.Out_XIntAttr(atNr, Count);
            AReport.Out_XIntAttr(atVid, lId);
            AReport.Out_XAttrEnd;
            AReport.Out_ArgTypes(fPrimaries);
            AReport.Out_Formula(fPredDef);
            AReport.Out_XElEnd(elDefPred);
         end;
         {$ENDIF}
      end;
   ikItmReconsidering:
      begin
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elReconsider);
         AReport.Out_XIntAttr(atNr, g.VarNbr+1);
         AReport.Out_XAttrEnd;
         {$ENDIF}
         LocBase := g.VarNbr;
         InFile.InWord;
         while InFile.Current.Kind <> ';' do
         begin
            inc(g.VarNbr); mizassert(2522,g.VarNbr<=MaxVarNbr);
            BoundVarNbr := 0;
            InFile.InWord;  // 'I'
            FixedVar[g.VarNbr].nIdent := InFile.Current.Nr;
            lExpPtr := LoadTerm;
            lTrm := lExpPtr^.Analyze;
            if lTrm^.TrmSort=ikTrmQua then lTrm := QuaTrmPtr(lTrm)^.TrmProper;
            FixedVar[g.VarNbr].nExp := false;
            FixedVar[g.VarNbr].nDef := lTrm;
            InFile.InWord;
         end;
```

```
            InFile.InWord;
            lTyp := ReadType;
            {$IFDEF ANALYZER_REPORT}
            for i := LocBase+1 to g.VarNbr do
            begin
               AReport.Out_TypeWithId(lTyp, FixedVar[i].nIdent);
               AReport.Out_Term(FixedVar[i].nDef);
            end;
            {$ENDIF}
            for i := LocBase+1 to g.VarNbr do
            begin
               FixedVar[i].nTyp := lTyp^.CopyType;
            end;
            lFrm := NewVerum;
            for i := LocBase+1 to g.VarNbr do
            begin
               { sa nie kopiowane i beda rozdysponowane razem z formula }
               { are not copied and will be distributed together with the formula }
               lFrm := NewConj(lFrm,NewQualFrm(FixedVar[i].nDef,TypPtr(lTyp^.CopyType)));
            end;
            dispose(lTyp,Done);
            Justify(0,0,lFrm);
            dispose(lFrm,Done);
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElEnd(elReconsider);
            {$ENDIF}
         end;
      ikItmChoice:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elConsider);
            AReport.Out_XIntAttr(atNr, g.VarNbr+1);
            AReport.Out_XAttrEnd;
            {$ENDIF}
            GetQualifiedList;
            ReadPropositions(lConditions);
            lFrm := xFormula(ConjugatePropositions(lConditions));
            Justify(0,0,lFrm);
            dispose(lFrm,Done);
            WriteQualified;
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_Propositions(lConditions);
            AReport.Out_XElEnd(elConsider);
            {$ENDIF}
            lConditions.Done;
         end;
      'E':
         begin
            RegularStatement(lFrm);
            dispose(lFrm,Done);
         end;
   else
   begin
      {$IFDEF MDEBUG}
      writeln(InfoFile,InFile.Current.Kind,'|');
      {$ENDIF}
      RunTimeError(2070);
   end;
   end;
   for i := lVarBase+1 to g.VarNbr do FixedVar[i].nSkelConstNr := 0;
```

```
end;
```

This code is used in chunk 25.
Defines:
`Statement`, used in chunks 78c, 88b, 97, 121, 129, 132, and 138.
Uses `AnalizeArgTypeList` 8b, `Analyze` 138, `ConjugatePropositions` 9b, `GetQualifiedList` 13b, `Justify` 96b, `ReadPropositions` 9a, `ReadSentence` 7a, `ReadTerm` 8a, `ReadType` 7b, `RegularStatement` 97, `WriteQualified` 14b, and `xFormula` 13a.

1.4 Properties

103a ⟨*Analyzer methods* 5a⟩+≡

```
{--- Start analysis of Properties ---}
⟨Change loci in property 103b⟩
⟨Change loci in sethood property 103c⟩
⟨Swap loci in type 104a⟩
⟨Parse predicate property 104b⟩
⟨Parse functor property 104c⟩
⟨Parse mode property 105a⟩
⟨Set visible (two) 105b⟩
⟨Set visible (one) 106a⟩
⟨Process properties 106b⟩
{--- End Properties ---}
```

This code is used in chunk 3c.

103b ⟨*Change loci in property* 103b⟩≡

```
var gVisible1,gVisible2,gFirstArg, gSecondArg: integer;

procedure ChangeLociInProperty(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
   with VarTrmPtr(fTrm)^ do
      case TrmSort of
         ikTrmBound: inc(VarNr,gBoundInc);
         ikTrmConstant:
            begin
               if VarNr = gFirstArg then
               begin TrmSort := ikTrmBound; VarNr := gBoundForFirst; exit end;
               if VarNr = gSecondArg then
               begin TrmSort := ikTrmBound; VarNr := gBoundForSecond; exit end;
            end;
         ikTrmIt:
            begin lTrm := fTrm; fTrm := NewVarTrm(ikTrmBound,gBoundForIt);
            dispose(lTrm,Done);
            end;
      end;
end;
```

This code is used in chunk 103a.
Defines:
`ChangeLociInProperty`, used in chunk 106b.
`gFirstArg`, used in chunks 104–106.
`gSecondArg`, used in chunks 104–106.
`gVisible1`, used in chunks 104–106.
`gVisible2`, used in chunks 104–106.

103c ⟨*Change loci in sethood property* 103c⟩≡

```
procedure ChangeLociInPropertySetHood(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
   with VarTrmPtr(fTrm)^ do
```

```
      case TrmSort of
         ikTrmBound: inc(VarNr,gBoundInc);
         ikTrmIt:
            begin
               lTrm := fTrm;
               fTrm := NewVarTrm(ikTrmBound,gBoundForIt);
               dispose(lTrm,Done);
            end;
      end;
  end;
```

This code is used in chunk 103a.
Defines:
ChangeLociInPropertySetHood, used in chunk 106b.

104a ⟨*Swap loci in type* 104a⟩≡

```
  procedure SwapLociInType(var fTrm: TrmPtr);
  begin
     with VarTrmPtr(fTrm)^ do
        case TrmSort of
           ikTrmConstant:
              begin
                 if VarNr = gFirstArg then begin VarNr := gSecondArg; exit end;
                 if VarNr = gSecondArg then begin VarNr := gFirstArg; exit end;
              end;
        else
        end;
  end;
```

This code is used in chunk 103a.
Defines:
SwapLociInType, used in chunk 106b.
Uses gFirstArg 103b and gSecondArg 103b.

104b ⟨*Parse predicate property* 104b⟩≡

```
  procedure PredProperty(fProp: integer);
  begin
     if RedefAntonym then
        case fProp of
           2: fProp := 3;
           3: fProp := 2;
           7: fProp := 8;
           8: fProp := 7;
        end;
     with gProperties do
     begin
        nFirstArg := gVisible1;
        nSecondArg := gVisible2;
        include(Properties,PropertyKind(fProp));
     end;
  end;
```

This code is used in chunk 103a.
Defines:
PredProperty, used in chunk 106b.
Uses gVisible1 103b and gVisible2 103b.

104c ⟨*Parse functor property* 104c⟩≡

```
  procedure FuncProperty(fProp: integer);
  begin
     with gProperties do
     begin
        nFirstArg := gVisible1;
        nSecondArg := gVisible2;
```

```
      include(Properties,PropertyKind(fProp));
    end;
  end;
```

This code is used in chunk 103a.
Defines:
FuncProperty, used in chunk 106b.
Uses gVisible1 103b and gVisible2 103b.

105a ⟨*Parse mode property* 105a⟩≡

```
  procedure ModeProperty(fProp: integer);
  begin
    with gProperties do
    begin
      nFirstArg := 0;
      nSecondArg := 0;
      include(Properties,PropertyKind(fProp));
    end;
  end;
```

This code is used in chunk 103a.
Defines:
ModeProperty, used in chunk 106b.

105b ⟨*Set visible (two)* 105b⟩≡

```
  procedure SetVisible2(ff: char);
  var
    lVisible: IntSequencePtr;
  begin
    gStatusOfProperties := 2;
    lVisible := nil;
    case ff of
      'R':
        with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
                                    lVisible := @Visible;
      'K':
        with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                                  lVisible := @Visible;
      else RunTimeError(2999);
    end;
    if lVisible^.fCount = 2 then
    begin
      gStatusOfProperties := 3;
      gVisible1 := lVisible^.fList^[0];
      gVisible2 := lVisible^.fList^[1];
      if (gVisible1 = 0) or (gVisible2 = 0) then exit;
      gFirstArg := LocusAsConst[gVisible1];
      gSecondArg := LocusAsConst[gVisible2];
      if StrictEqTyp(FixedVar[gFirstArg].nTyp,FixedVar[gSecondArg].nTyp) then
      begin
        gStatusOfProperties := 4;
      { Przy absolutnej permisywnosci bedzie mozna opuscic
        warunek. Trzeba jednak wymagac aby zalozenie, takze
        ukryte, tzn. koniunkcja negacji dozorow, jezeli
        brak "otherwise" byla symetryczna.
      }
      { If you are absolutely permissive, you will be able to skip
        the condition. However, it is necessary to require that the assumption, also
        hidden, i.e. the conjunction of the negation of supervisions, if
        there is no "otherwise", is symmetric.
      }
        if gNonPermissive then
          if (gDefiniens = nil) or (gDefiniens^.nOtherwise <> nil) then
```

```
                gStatusOfProperties := 1;
          end;
       end;
    end;
```

This code is used in chunk 103a.
Defines:
SetVisible2, used in chunk 106b.
Uses gFirstArg 103b, gSecondArg 103b, gVisible1 103b, and gVisible2 103b.

106a ⟨*Set visible (one)* 106a⟩≡

```
procedure SetVisible1(ff: char);
begin
   gStatusOfProperties := 2;
   if ff <> 'K' then RunTimeError(2999);
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
   if Visible.fCount = 1 then
   begin
      gStatusOfProperties := 3;
      gVisible1 := Visible.fList^[0]; gVisible2 := 0;
      if gVisible1 = 0 then exit;
      gFirstArg := LocusAsConst[gVisible1];
      gSecondArg := 0;
      gStatusOfProperties := 4;
      if gNonPermissive then
         if (gDefiniens = nil) or (gDefiniens^.nOtherwise <> nil) then
            gStatusOfProperties := 1;
   end;
end;
```

This code is used in chunk 103a.
Defines:
SetVisible1, used in chunk 106b.
Uses gFirstArg 103b, gSecondArg 103b, gVisible1 103b, and gVisible2 103b.

106b ⟨*Process properties* 106b⟩≡

```
procedure ProcessProperties(ff: char);
var
   lPropCond: FrmPtr;
   function TheFormula(fBoundInc,fIt,fBound1,fBound2: integer): FrmPtr;
   var
      lFrm,lOth,llFrm: FrmPtr;
      z: integer;
   begin
      if gDefiniens = nil then
      begin
         { Jezeli gDefiniens = nil, to to musi byc redefinicja, inaczej
           jedyna formula, ktora mozna wyprodukowac jest bledna.
         }
         { If gDefiniens = nil, then this must be a redefinition, otherwise
           the only formula that can be produced is incorrect. }
         if gRedef then
            with Notat[noPredicate], PatternPtr(Items^[Count+fExtCount-1])^ do
            lFrm := NewPredFrm(ikFrmPred,gWhichOne,LociList(fPrimTypes.Count),Count+fExtCount)
         else lFrm := NewIncorFrm;
      end
      else
         with gDefiniens^ do
      begin
         mizassert(2601,nOtherwise <> nil);
         lFrm := NewVerum; lOth := NewVerum;
         with nPartialDefinientia do
            for z := 0 to Count-1 do
```

```
            with PartDefPtr(Items^[z])^ do
         begin
            lOth := NewConj(lOth,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
            case DefSort of
               'm': llFrm := FrmPtr(nPartDefiniens)^.CopyFormula;
               'e': llFrm := NewEqFrm(NewItTrm,CopyTerm(TrmPtr(nPartDefiniens)));
            else RunTimeError(2511);
            end;
            lFrm := NewConj(lFrm,NewImpl(FrmPtr(nGuard)^.CopyFormula,llFrm));
         end;
      case DefSort of
         'm': llFrm := FrmPtr(gDefiniens^.nOtherwise)^.CopyFormula;
         'e': llFrm := NewEqFrm(NewItTrm,CopyTerm(TrmPtr(gDefiniens^.nOtherwise)));
      else RunTimeError(2512);
      end;
      lFrm := NewConj(lFrm,NewImpl(lOth,llFrm));
   end;
   gBoundInc := fBoundInc;
   gBoundForFirst := fBound1; gBoundForSecond := fBound2;
   gBoundForIt := fIt;
   WithinFormula(lFrm,ChangeLociInProperty);
   TheFormula := lFrm;
end;

function Reflexivity:FrmPtr;
begin
   if not RedefAntonym then
      Reflexivity :=
         NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                  TheFormula(1,1,1,1))
   else Reflexivity :=
      NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
               NewNegDis(TheFormula(1,1,1,1)));
end;

function Irreflexivity:FrmPtr;
begin
   if not RedefAntonym then
      Irreflexivity :=
         NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                  NewNegDis(TheFormula(1,1,1,1)))
   else Irreflexivity :=
      NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
               TheFormula(1,1,1,1));
end;

function Symmetry: FrmPtr;
begin
   if not RedefAntonym then Symmetry :=
      NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
               NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                        NewImpl(TheFormula(2,1,1,2),
                                TheFormula(2,1,2,1))))
   else Symmetry :=
      NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
               NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                        NewImpl(NewNegDis(TheFormula(2,1,1,2)),
                                NewNegDis(TheFormula(2,1,2,1)))));
end;
```

```
    function Asymmetry: FrmPtr;
    begin
       if not RedefAntonym then Asymmetry :=
          NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                   NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                            NewImpl(TheFormula(2,1,1,2),
                                    NewNegDis(TheFormula(2,1,2,1)))))
       else Asymmetry :=
          NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                   NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                            NewImpl(NewNegDis(TheFormula(2,1,1,2)),
                                    TheFormula(2,1,2,1))));
    end;

    function Connectedness: FrmPtr;
    begin
       if not RedefAntonym then Connectedness :=
          NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                   NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                            NewImpl(NewNegDis(TheFormula(2,1,1,2)),
                                    TheFormula(2,1,2,1))))
       else Connectedness :=
          NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                   NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                            NewImpl(TheFormula(2,1,1,2),
                                    NewNegDis(TheFormula(2,1,2,1)))));
    end;

  var
    gPropPos: Position;

    function Commutativity: FrmPtr;
    var lTrm1,lTrm2:TrmPtr; lLength:integer;
    lTypPtr: TypPtr;
    begin
       with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                               lLength := fPrimTypes.Count;
       { Konieczna jest kontrola typu: czy przy przestawieniu argumentow
         sie przypadkiem typ ItTyp nie zmienia.
         Informacja od Grzegorza.
       }
       { It is necessary to check the type: whether the ItTyp type changes
         when the arguments are changed. Information from Grzegorz. }
       lTypPtr := ItTyp^.CopyType;
       lTypPtr^.WithinType(SwapLociInType);
       if not EqTyp(ItTyp,lTypPtr) then Error(gPropPos,84);
       dispose(lTypPtr,Done);

       if gDefiniens = nil then
          { Wyjatek z ogolnych regul. Uproszczona formula ! }
          {Exception from general regulations. Simplified formula! }
          if gRedef then
          begin
             gBoundInc := 2;
             gBoundForFirst := 1; gBoundForSecond := 2;
             lTrm1 := NewFuncTrm(gWhichOne,LociList(lLength));
             WithinTerm(lTrm1,ChangeLociInProperty);
             gBoundForFirst := 2; gBoundForSecond := 1;
             lTrm2 := NewFuncTrm(gWhichOne,LociList(lLength));
             WithinTerm(lTrm2,ChangeLociInProperty);
```

```
            Commutativity :=
               NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                        NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                                 NewEqFrm(lTrm1,lTrm2)));
         end
         else Commutativity := NewIncorFrm
         else
            Commutativity :=
               NewUniv(ItTyp^.CopyType,
                       NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                                NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                                         NewImpl(TheFormula(3,1,2,3),TheFormula(3,1,3,2)))));
   end;

   function Idempotence: FrmPtr;
   var lVisible1,lFirstArg:Integer;
   begin
      with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                              lVisible1 := Visible.fList^[0];
      lFirstArg := LocusAsConst[lVisible1];
      if not ItTyp^.IsWiderThan(FixedVar[lFirstArg].nTyp^.CopyType) then
      begin
         Idempotence := NewIncorFrm;
         Error(gPropPos,78);
         exit;
      end;
      if gRedef then
      begin
         Idempotence := NewIncorFrm;
         Error(gPropPos,89);
      end
      else
         if gDefiniens = nil then Idempotence := NewIncorFrm
         else
            Idempotence := NewUnivI(FixedVar[gFirstArg].nIdent,
                                    FixedVar[lFirstArg].nTyp^.CopyType,TheFormula(1,1,1,1));
   end;

   function Involutiveness: FrmPtr;
   var lVisible1,lFirstArg:Integer;
   begin
      with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                              lVisible1 := Visible.fList^[0];
      lFirstArg := LocusAsConst[lVisible1];
      if not EqTyp(ItTyp,FixedVar[lFirstArg].nTyp) then
      begin
         Involutiveness := NewIncorFrm;
         Error(gPropPos,85);
         exit;
      end;
      if gRedef then
      begin
         Involutiveness := NewIncorFrm;
         Error(gPropPos,89);
      end
      else
         if gDefiniens = nil then Involutiveness := NewIncorFrm
         else
            Involutiveness := NewUniv(ItTyp^.CopyType,NewUniv(ItTyp^.CopyType,
                                                              NewImpl(TheFormula(2,1,2,0),TheFormula(2,2,1,0))));
```

```
end;

function Projectivity: FrmPtr;
var lVisible1,lFirstArg:Integer;
lTyp:TypPtr;
begin
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
                              lVisible1 := Visible.fList^[0];
   lFirstArg := LocusAsConst[lVisible1];
   lTyp := ItTyp^.CopyType;
   if lTyp^.TypSort <> ikError then
      lTyp := FixedVar[lFirstArg].nTyp^.WideningOf(lTyp);
   if (lTyp = nil) or (lTyp^.TypSort = ikError) or
         not lTyp^.EqRadices(FixedVar[lFirstArg].nTyp) or
         not FixedVar[lFirstArg].nTyp^.LowerCluster^.IsSubsetOf(lTyp^.UpperCluster,EqAttr)
   then
   begin
      Projectivity := NewIncorFrm;
      Error(gPropPos,85);
      if lTyp <> nil then dispose(lTyp,Done);
      exit;
   end;
   dispose(lTyp,Done);
   if gRedef then
   begin
      Projectivity := NewIncorFrm;
      Error(gPropPos,89);
   end
   else
      if gDefiniens = nil then Projectivity := NewIncorFrm
      else
         Projectivity := NewUniv(ItTyp^.CopyType,NewUniv(FixedVar[lFirstArg].nTyp^.CopyType,
                                                         NewImpl(TheFormula(2,1,2,0),TheFormula(2,1,1,0))));
end;

function Associativity: FrmPtr;
var lTrm1,lTrm2:TrmPtr;
lLength:integer;
begin
   ErrImm(77);
   Associativity := NewIncorFrm;
   exit;
   with Notat[noFunctor], PatternPtr(Items^[Count+fExtCount-1])^ do
   begin
      lLength := fPrimTypes.Count;
   end;
   if (gFirstArg = 0) or (gSecondArg = 0) then
   begin
      Associativity := NewIncorFrm;
      Error(gPropPos,85);
      exit;
   end;
   if not EqTyp(ItTyp,FixedVar[gFirstArg].nTyp) and
         not EqTyp(ItTyp,FixedVar[gSecondArg].nTyp)  then
   begin
      Associativity := NewIncorFrm;
      Error(gPropPos,85);
      exit;
   end;
   if gDefiniens = nil then
```

```
      if gRedef then
      begin gBoundInc := 2;
      gBoundForFirst := 1; gBoundForSecond := 2;
      lTrm1 := NewFuncTrm(gWhichOne,LociList(lLength));
      WithinTerm(lTrm1,ChangeLociInProperty);
      gBoundForFirst := 2; gBoundForSecond := 1;
      lTrm2 := NewFuncTrm(gWhichOne,LociList(lLength));
      WithinTerm(lTrm2,ChangeLociInProperty);
      Associativity :=
         NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                  NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                           NewEqFrm(lTrm1,lTrm2)));
      //for x,x,z st holds F(F(x,y),z) = F(x,F(y,z))
      end
      else Associativity := NewIncorFrm
      else
         Associativity :=
            NewUniv(ItTyp^.CopyType,
                    NewUnivI(FixedVar[gFirstArg].nIdent,TypPtr(FixedVar[gFirstArg].nTyp^.CopyType),
                             NewUnivI(FixedVar[gSecondArg].nIdent,TypPtr(FixedVar[gSecondArg].nTyp^.CopyType),
                                      NewImpl(TheFormula(3,1,2,3),TheFormula(3,1,3,2)))));

   // function TheFormula(fBoundInc,fIt,fBound1,fBound2:integer):FrmPtr;
   //for it,xy,yz,x,x,z st P[it,xy,z] & P[xy,x,y] & P[yz,y,z]
   //  holds P[it,x,yz]

end;

function Transitivity: FrmPtr;
begin
   ErrImm(77);
   Transitivity := NewIncorFrm;
end;

function Sethood: FrmPtr;
var
   lFrm,lOth,llFrm: FrmPtr;
   z: integer;
begin
   if gDefiniens = nil then
   begin
      lFrm := NewIncorFrm;
   end
   else
      with gDefiniens^ do
   begin
      gBoundInc := 2;
      gBoundForFirst := 0; gBoundForSecond := 0;
      gBoundForIt := 2;
      if nPartialDefinientia.Count = 0 then
      begin
         mizassert(2591,nOtherwise <> nil);
         lFrm := FrmPtr(gDefiniens^.nOtherwise)^.CopyFormula;
         WithinFormula(lFrm, ChangeLociInPropertySetHood);
         lFrm := NewExis(NewStandardTyp(ikTypMode,NewEmptyCluster,NewEmptyCluster,
                                        gBuiltIn[rqSetMode],nil),
                         NewUniv(ItTyp^.CopyType,
                                 NewImpl(lFrm,
                                         NewPredFrm(ikFrmPred,gBuiltIn[rqBelongsTo],
                                                    NewTrmList(NewVarTrm(ikTrmBound,2),
```

```
                                                                NewTrmList(NewVarTrm(ikTrmBound,1),nil)),0))));
          end
          else
          begin
             mizassert(2592,nOtherwise <> nil);
             lFrm := nil; lOth := NewVerum;
             with nPartialDefinientia do
                for z := 0 to Count-1 do
                   with PartDefPtr(Items^[z])^ do
                begin
                   lOth := NewConj(lOth,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
                   llFrm := FrmPtr(nPartDefiniens)^.CopyFormula;
                   WithinFormula(llFrm,ChangeLociInProperty);
                   llFrm := NewExis(NewStandardTyp(ikTypMode,NewEmptyCluster,NewEmptyCluster,
                                                   gBuiltIn[rqSetMode],nil),
                                    NewUniv(ItTyp^.CopyType,
                                            NewImpl(llFrm,
                                                    NewPredFrm(ikFrmPred,gBuiltIn[rqBelongsTo],
                                                               NewTrmList(NewVarTrm(ikTrmBound,2),
                                                                          NewTrmList(NewVarTrm(ikTrmBound,1),nil))
                   llFrm := NewConj(FrmPtr(nGuard)^.CopyFormula,llFrm);
                   if lFrm = nil
                   then lFrm := llFrm
                   else lFrm := NewDisj(lFrm,NewConj(FrmPtr(nGuard)^.CopyFormula,llFrm));
                end;
             llFrm := FrmPtr(gDefiniens^.nOtherwise)^.CopyFormula;
             WithinFormula(llFrm,ChangeLociInProperty);
             llFrm := NewExis(NewStandardTyp(ikTypMode,NewEmptyCluster,NewEmptyCluster,
                                             gBuiltIn[rqSetMode],nil),
                              NewUniv(ItTyp^.CopyType,
                                      NewImpl(llFrm,
                                              NewPredFrm(ikFrmPred,gBuiltIn[rqBelongsTo],
                                                         NewTrmList(NewVarTrm(ikTrmBound,2),
                                                                    NewTrmList(NewVarTrm(ikTrmBound,1),nil)),0))))
             lFrm := NewDisj(lFrm,NewConj(lOth,llFrm));
          end;
       end;
       SetHood := lFrm;
    end;

 begin  {--- ProcessProperties ---}
    while InFile.Current.Kind = 'X' do
    begin
       {$IFDEF ANALYZER_REPORT}
       AReport.Out_XElStart0(elJustifiedProperty);
       AReport.Out_XEl1(Prop2XmlElem[ PropertyKind(InFile.Current.Nr)]);
       {$ENDIF}
       gStatusOfProperties := 0;
       case ff of
          'R','K':
             if InFile.Current.Nr in [1..9] then SetVisible2(ff) else
                if InFile.Current.Nr in [10,11] then SetVisible1(ff)
                else if InFile.Current.Nr in [12] then
                   gStatusOfProperties := 1;
          'M':
             if InFile.Current.Nr in [12] then
             begin
                gVisible1 := 0;
                gVisible2 := 0;
                gFirstArg := 0;
```

```
                gSecondArg := 0;
                if gRedef then
                   gStatusOfProperties := 5
                else gStatusOfProperties := 1;
             end
             else;
          end;
          gPropertiesOcc := true;
          case gStatusOfProperties of
             0:
                begin
                   lPropCond := NewIncorFrm;
                   InFile.InPos(CurPos);
                end;
             1:
                begin InFile.InPos(CurPos); gPropPos := CurPos;
                case InFile.Current.Nr of
                   0: lPropCond := NewIncorFrm;
                   1:  begin PredProperty(1);  lPropCond := Symmetry end;
                   2:  begin PredProperty(2);  lPropCond := Reflexivity end;
                   3:  begin PredProperty(3);  lPropCond := Irreflexivity end;
                   4:  begin FuncProperty(4);  lPropCond := Associativity end;
                   5:  begin PredProperty(5);  lPropCond := Transitivity end;
                   6:  begin FuncProperty(6);  lPropCond := Commutativity end;
                   7:  begin PredProperty(7);  lPropCond := Connectedness end;
                   8:  begin PredProperty(8);  lPropCond := Asymmetry end;
                   9:  begin FuncProperty(9);  lPropCond := Idempotence end;
                   10: begin FuncProperty(10); lPropCond := Involutiveness end;
                   11: begin FuncProperty(11); lPropCond := Projectivity end;
                   12: begin ModeProperty(12); lPropCond := SetHood end;
                else RunTimeError(2013);
                end;
                end;
          else
          begin lPropCond := NewIncorFrm;
          InFile.InPos(CurPos);
          case gStatusOfProperties of
             2: if InFile.Current.Nr in [1,2,3,5,7,8] then ErrImm(81) else
                   if InFile.Current.Nr in [6,9] then ErrImm(82) else
                      if InFile.Current.Nr in [10,11] then ErrImm(83) else RuntimeError(2999);
             3: ErrImm(79);
             4: ErrImm(80);
             5: ErrImm(77);
          end;
          end;
          end;
          Justify(0,0,lPropCond);
          dispose(lPropCond,Done);
          {$IFDEF ANALYZER_REPORT}
          AReport.Out_XElEnd(elJustifiedProperty);
          {$ENDIF}
       end;
    end;
```

This code is used in chunk 103a.
Defines:
ProcessProperties, used in chunk 121.
Uses ChangeLociInProperty 103b, ChangeLociInPropertySetHood 103c, FuncProperty 104c, gFirstArg 103b, gSecondArg 103b, gVisible1 103b, gVisible2 103b, Justify 96b, LociList 16a, ModeProperty 105a, PredProperty 104b, SetVisible1 106a, SetVisible2 105b, and SwapLociInType 104a.

1.5 Parsing Definitions

114a ⟨*Analyzer methods* 5a⟩+≡
⟨*Parse parametrization* 114b⟩
⟨*Determine meaning* 115⟩
⟨*Determine meaning for equation* 116a⟩
⟨*Parse "CC" Formal arguments* 116b⟩
⟨*Parse "BB" Formal arguments* 117a⟩
⟨*Change declared constants to loci* 117b⟩
⟨*Create Definientia* 117c⟩
⟨*Analyze definitional theorems* 119⟩
⟨*Parse definitions* 121⟩
⟨*Round up item* 128⟩
⟨*Parse a registration* 129⟩
⟨*Parse notation* 132⟩
⟨*Parse a scheme block* 134⟩
⟨*Analyze reduction-like theorem* 135a⟩
⟨*Parse a theorem* 135b⟩
⟨*Parse section* 136a⟩
⟨*Parse cancelled item* 136b⟩
⟨*Load SGN environment file(?)* 137a⟩
⟨*Dispose analyze* 137b⟩
⟨*Analyze* 138⟩
This code is used in chunk 3c.

114b ⟨*Parse parametrization* 114b⟩≡

```
procedure Parametrization;
var
   i,lNbr: integer;
   lTyp: TypPtr;
begin
   gFixedBase := g.VarNbr;
   InFile.InPos(CurPos);
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart(elLet);
   AReport.Out_XIntAttr(atNr, g.VarNbr+1);
   AReport.Out_XAttrEnd;
   {$ENDIF}
   InFile.InWord;
   while InFile.Current.Kind='Q' do
   begin
      lNbr := g.VarNbr;
      inc(g.VarNbr,InFile.Current.Nr);
      if g.VarNbr-gDefBase > gMaxArgNbr then
         OverflowError(937);
      for i := 1 to InFile.Current.Nr do
      begin
         InFile.InWord; // 'I'
         FixedVar[lNbr+i].nIdent := InFile.Current.Nr;
      end;
      gFraenkelTermAllowed := false;
      lTyp := ReadType;
      gFraenkelTermAllowed := true;
      for i := lNbr+1 to g.VarNbr do
      begin
         if lTyp^.TypSort<>IkError then
         begin
            if i=g.VarNbr then
               FixedVar[i].nTyp := lTyp
            else FixedVar[i].nTyp := lTyp^.CopyType
```

```
            end
            else FixedVar[i].nTyp := NewIncorTyp;
            FixedVar[i].nExp := false;
         end;
         //   dispose(lTyp,Done);
         InFile.InWord;
      end;
      WriteQualified;
      ParamDecl(gFixedBase);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elLet);
      {$ENDIF}
   end;
```

This code is used in chunk 114a.
Defines:
Parametrization, used in chunks 121, 129, and 132.
Uses gFixedBase 12c, gMaxArgNbr 8b, ParamDecl 15b, ReadType 7b, and WriteQualified 14b.

115 ⟨*Determine meaning* 115⟩≡

```
   function Meaning(fDef: DefPtr;
                    Definiendum: FrmPtr): FrmPtr;
   var
      dFrm2,dFrm: FrmPtr;
      z: integer;
   begin
      with fDef^ do
      begin
         dFrm := NewVerum;
         mizassert(2597,DefSort='m');
         if nOtherwise <> nil then dFrm2 := NewVerum;
         with nPartialDefinientia do
            for z := 0 to Count-1 do
               with PartDefPtr(Items^[z])^ do
            begin
               if fDef^.nOtherWise<>nil then
                  dFrm2 := NewConj(dFrm2,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
               dFrm := NewConj(dFrm,
                               NewImpl(FrmPtr(nGuard),
                                       NewBicond(Definiendum^.CopyFormula,FrmPtr(nPartDefiniens))));
            end;
         if nOtherWise<>nil then
            dFrm := NewConj(dFrm,
                            NewImpl(dFrm2,NewBicond(Definiendum^.CopyFormula,FrmPtr(nOtherWise))));
      end;
      dispose(Definiendum,Done);
      with fDef^.nPartialDefinientia do
      begin
         for z := 0 to Count-1 do dispose(PartDefPtr(Items^[z]));
         DeleteAll; Done;
      end;
      dispose(fDef);
      WithInFormula(dFrm,ChangeDeclConstToBound);
      Meaning := dFrm;
   end;
```

This code is used in chunk 114a.
Defines:
Meaning, used in chunk 119.
Uses ChangeDeclConstToBound 87a.

116a ⟨*Determine meaning for equation* 116a⟩≡

```
function MeaningEq(fDef: DefPtr;
                   Definiendum: TrmPtr): FrmPtr;
var
   dFrm2,dFrm: FrmPtr;
   z: integer;
begin
   with fDef^ do
   begin
      dFrm := NewVerum;
      mizassert(2598,DefSort='e');
      if nOtherwise <> nil then dFrm2 := NewVerum;
      with nPartialDefinientia do
         for z := 0 to Count-1 do
            with PartDefPtr(Items^[z])^ do
         begin
            if fDef^.nOtherWise<>nil then
               dFrm2 := NewConj(dFrm2,NewNegDis(FrmPtr(nGuard)^.CopyFormula));
            dFrm := NewConj(dFrm,
                            NewImpl(FrmPtr(nGuard)^.CopyFormula,
                                    NewEqFrm(CopyTerm(Definiendum),CopyTerm(TrmPtr(nPartDefiniens)))));
         end;
      if nOtherWise<>nil then
         dFrm := NewConj(dFrm,
                         NewImpl(dFrm2,
                                 NewEqFrm(CopyTerm(Definiendum),CopyTerm(TrmPtr(nOtherWise)))));
   end;
   dispose(fDef,Done);
   WithInFormula(dFrm,ChangeDeclConstToBound);
   MeaningEq := dFrm;
end;
```

This code is used in chunk 114a.
Defines:
MeaningEq, used in chunk 119.
Uses ChangeDeclConstToBound 87a.

116b ⟨*Parse "CC" Formal arguments* 116b⟩≡

```
function CC_FormalArgs: TrmList;
var
   lTrmList: TrmList;
   Previous: ^TrmList;
   k: integer;
begin
   Previous := addr(lTrmList);
   for k := g.DemBase+1 to g.VarNbr do
      if FixedVar[k].nSkelConstNr<>0 then
      begin
         new(Previous^);
         Previous^^.XTrmPtr := NewVarTrm(ikTrmConstant,k);
         Previous := addr(Previous^^.NextTrm);
      end;
   Previous^ := nil;
   CC_FormalArgs := lTrmList;
end;
```

This code is used in chunk 114a.
Defines:
CC_FormalArgs, used in chunks 117a and 119.

117a ⟨*Parse "BB" Formal arguments* 117a⟩≡
```
function BB_FormalArgs: TrmList;
var
   ltl: TrmList;
begin
   ltl := CC_FormalArgs;
   BB_FormalArgs := ltl;
   while ltl<>nil do with ltl^ do
   begin WithInTerm(XTrmPtr,ChangeDeclConstToBound); ltl := NextTrm end;
end;
```
This code is used in chunk 114a.
Defines:
BB_FormalArgs, used in chunk 119.
Uses CC_FormalArgs 116b and ChangeDeclConstToBound 87a.

117b ⟨*Change declared constants to loci* 117b⟩≡
```
procedure ChangeDeclConstToLoci(var fTrm: TrmPtr);
var
   lTrm: TrmPtr;
begin
   with VarTrmPtr(fTrm)^ do
      case  TrmSort of
         ikTrmConstant:
            if (VarNr>g.DemBase) and (FixedVar[VarNr].nSkelConstNr<>0) then
            begin TrmSort := ikTrmLocus; VarNr := FixedVar[VarNr].nSkelConstNr end;
         ikTrmIt:
            begin lTrm := fTrm; fTrm := NewVarTrm(ikTrmLocus,g.GenCount);
            dispose(lTrm,Done);
            end;
      end;
end;
```
This code is used in chunk 114a.
Defines:
ChangeDeclConstToLoci, used in chunk 117c.

117c ⟨*Create Definientia* 117c⟩≡
```
var gDefThNr: integer = 0;   // count (also canceled) deftheorems
procedure CreateDefinientia;
   procedure CreateDefiniens(Item: DefNodePtr);
   var
      aFrm: FrmPtr;
      lEntry: RsnEntry;
      lPartialPart: MCollection;
      lOtherWise,lPartDef: PObject;
      lGuard: FrmPtr;
      lKind: Char;
      lNr,lLabId,z: integer;
   begin
      with Item^ do
      begin
         // the deftheorem will be created also for canceled
         if (DDef = nil) and (nConstructor.Kind = ':') then inc(gDefThNr);
         if DDef <> nil then
         begin
            g.GenCount := SkIt; lLabId := SkLabId;
            inc(gDefThNr);  // the deftheorem will be created also for canceled
            { Poniewaz bardziej dokladne informacje sa potrzebne dla
              konstrukcji twierdzenia definicyjnego, jest to chyba dobre
              miejsce, zeby je tutaj zmienic. }
            { Since more detailed information is needed for
              the construction of the definitional theorem, this is probably a good
```

```
  place to change it here. }
lKind := nConstructor.Kind;
lNr := nConstructor.Nr;
case lKind of
   'M','R','V','K':
      with ConstrPtr(Constr[ ConstructorKind(lKind)].Items^[lNr])^ do
                  if fWhichConstrNr<>0 then lNr := fWhichConstrNr;
   //        { dla funktorow nie tworzymy definiensow }
             { we do not create definitions for functors }
   ':': exit;
end;
{ ------------------------ }
aFrm := NewVerum; lEntry := nPrefix;
while LEntry <> nil do with LEntry^ do
begin
   if Form='A' then aFrm := NewConj(DSnt^.CopyFormula,aFrm);
   LEntry := PreviousEntry;
end;
WithInFormula(aFrm,ChangeDeclConstToLoci);
with DDef^ do
begin lPartialPart.Init(nPartialDefinientia.Count,0);
with nPartialDefinientia do
   for z := 0 to Count-1 do
      with PartDefPtr(Items^[z])^ do
   begin
      case DefSort of
         'm':
            begin
               lPartDef := FrmPtr(nPartDefiniens)^.CopyFormula;
               WithInFormula(FrmPtr(lPartDef),ChangeDeclConstToLoci);
            end;
         'e':
            begin
               lPartDef := CopyTerm(TrmPtr(nPartDefiniens));
               WithInTerm(TrmPtr(lPartDef),ChangeDeclConstToLoci);
            end;
      else RunTimeError(2515);
      end;
      lGuard := FrmPtr(nGuard)^.CopyFormula;
      WithInFormula(lGuard,ChangeDeclConstToLoci);
      lPartialPart.Insert(new(PartDefPtr, Init(lPartDef,lGuard)));
   end;
lOtherWise := nil;
if nOtherWise <> nil then
begin
   case DefSort of
      'm':
         begin
            lOtherWise := FrmPtr(nOtherWise)^.CopyFormula;
            WithInFormula(FrmPtr(lOtherWise),ChangeDeclConstToLoci);
         end;
      'e':
         begin
            lOtherWise := CopyTerm(TrmPtr(nOtherWise));
            WithInTerm(TrmPtr(lOtherWise),ChangeDeclConstToLoci);
         end;
   else RunTimeError(2516);
   end;
end;
Definientia.Insert(
```

```
                new(DefiniensPtr,
                    Init(lKind,lNr,gDefThNr,lLabId,ArticleID,nPrimaryList,nEssentials,aFrm,
                         new(DefPtr,Init(DefSort,lPartialPart,lOtherwise)))));
             {
                              EqDefinientia.Insert(
                              new(DefiniensPtr,
                              Init(lKind,lNr,gDefThNr,lLabId,ArticleID,nPrimaryList,nEssentials,aFrm,
                              new(DefPtr,Init(DefSort,lPartialPart,lOtherwise)))));
  }
             end;
          end;
       end;
    end;
  var
    z,lDefBase: integer;
  begin
    lDefBase := Definientia.Count;
    with DefinitionList do
       for z := 0 to Count-1 do
          CreateDefiniens(DefNodePtr(Items^[z]));
    with Definientia do
       for z := lDefBase to Count-1 do
          with DefiniensPtr(Items^[z])^ do
             if Definiens<>nil then
             begin
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_Definiens(DefiniensPtr(Items^[z])^, z+1);
                {$ENDIF}
             end;
  end;
```

This code is used in chunk 114a.
Defines:
  `CreateDefinientia`, used in chunk 121.
  `gDefThNr`, never used.
Uses `ChangeDeclConstToLoci` 117b, `DefinitionList` 29a, and `DefNodePtr` 4b.

119 ⟨*Analyze definitional theorems* 119⟩≡

```
  procedure DefinitionalTheorems;
    procedure ProcessDefinition(Item:DefNodePtr);
    var
       lSkDef: DefPtr;
       lFrm,lFrm1: FrmPtr;
       nAttrNr: integer;
       lArgs,A :TrmList;
       NewType,lTyp: TypPtr;
       Sample: TrmPtr;
       ldefEntry: RSNENTRY;
       lDefProp: PropositionPtr;
    begin
       with Item^ do
       begin
          lDefProp := nil;
          g.VarNbr := SkVarNbr;
          if DDef<>nil then
          begin
             lSkDef := DDef;
             g.GenCount := SkIt;
             case nConstructor.Kind of
                'R': lFrm := Meaning(lSkDef,NewPredFrm(ikFrmPred,nConstructor.Nr,CC_FormalArgs,0));
                'V':
                   if  nConstructor.Nr = 0 then
```

```
                lFrm := NewInCorFrm
             else
             begin
                lFrm1 := NewPredFrm(ikFrmAttr,nConstructor.Nr,CC_FormalArgs,0);
                AdjustAttrFrm(PredFrmPtr(lFrm1),nAttrNr,A);
                lFrm := Meaning(lSkDef,NewPredFrm(ikFrmAttr,nAttrNr,CopyTermList(A),0));
                dispose(lFrm1,Done);
             end;
          'K':
             begin Sample := NewFuncTrm(nConstructor.Nr,CC_FormalArgs);
             if nMeansOccurs = 'e' then
             begin dec(g.GenCount);
             lFrm := MeaningEq(lSkDef,Sample);
             inc(g.GenCount);
             end
             else
             begin
                lArgs := BB_FormalArgs;
                lTyp := ConstrTypPtr(Constr[coFunctor].Items^[nConstructor.Nr])^.fConstrTyp^.InstTyp(lArgs);
                DisposeTrmList(lArgs);
                lFrm := NewUniv(lTyp,Meaning(lSkDef,NewEqFrm(NewItTrm,Sample)));
             end;
             end;
          'M':
             with ConstrTypPtr(Constr[coMode].Items^[nConstructor.Nr])^ do
          begin
             NewType :=
                NewStandardTyp(ikTypMode,NewEmptyCluster,
                               InstCluster(fConstrTyp^.UpperCluster,CC_FormalArgs),
                               nConstructor.Nr,CC_FormalArgs);
             lArgs := BB_FormalArgs;
             lTyp := fConstrTyp^.InstTyp(lArgs);
             DisposeTrmList(lArgs);
             lFrm := NewUniv(lTyp,Meaning(lSkDef,NewQualFrm(NewItTrm,NewType)));
          end;
       end;
       ldefEntry := nPrefix;
       while ldefEntry <> nil do with ldefEntry^ do
       begin
          case Form of
             'D': lFrm := NewUnivList(SkList,SkIdents,lFrm);
             'A': lFrm := NewImpl(SkSnt^.CopyFormula,lFrm);
          else RunTimeError(2010);
          end;
          ldefEntry := PreviousEntry;
       end;
       lDefProp := new(PropositionPtr, Init(SkId, SkLabId, lFrm, CurPos));
    end
    else if nConstructor.Kind = ':'
              { Przetwarzanie "canceled" } { Processing "canceled" }
    then lDefProp := new(PropositionPtr, Init(0, 0, NewVerum, CurPos));
    if Assigned(lDefProp) then
    begin
       {$IFDEF ANALYZER_REPORT}
       AReport.Out_XElStart(elDefTheorem);
       with AReport, nConstructor do
                      if Kind in [ 'K','M','R','V'] then
                      begin
                         Out_XAttr(atConstrKind, Kind);
                         Out_XIntAttr(atConstrNr, Transf(ConstructorKind(Kind), Nr));
```

```
                        end;
                AReport.Out_XAttrEnd;
                AReport.Out_Proposition(lDefProp);
                AReport.Out_XElEnd(elDefTheorem);
                {$ENDIF}
                dispose(lDefProp, Done);
             end;
          end;
       end;
    var
       z: integer;
    begin
       with DefinitionList do
          for z := 0 to Count-1 do
             ProcessDefinition(DefNodePtr(Items^[z]));
       DefinitionList.Done;
    end;
```

This code is used in chunk 114a.
Defines:
DefinitionalTheorems, used in chunk 121.
Uses BB_FormalArgs 117a, CC_FormalArgs 116b, DefinitionList 29a, DefNodePtr 4b, Meaning 115, MeaningEq 116a, and NewUnivList 90.

121 ⟨*Parse definitions* 121⟩≡

```
    procedure Definition;
    var
       lDeclBase,lVarBase,i,pVarNbr: integer;
       lConditions: MCollection;
       lFrm: FrmPtr;
       lPos: Position;
       lEntry: RSNENTRY;
       lNotatExtCount: array[NotationKind] of integer;
       nk: NotationKind;
       {$IFDEF ANALYZER_REPORT}
       procedure Do_Patterns;
       var
          k: integer;
          nk1: NotationKind;
       begin
          for nk1 := Low(NotationKind) to High(NotationKind) do
             with Notat[nk1] do
                for k := Count + lNotatExtCount[nk1] to Count + fExtCount - 1 do
                   AReport.Out_Pattern(Items^[k], k+1);
       end;
       {$ENDIF}
    begin
       InFile.InPos(CurPos);
       gDefiniendumArgs := nil;
       d := g;
       D.LocPredNbr := LocPredDef.Count;
       D.LocFuncNbr := LocFuncDef.Count;
       OpenDef;
       {$IFDEF ANALYZER_REPORT}
       AReport.Out_XElStart(elDefinitionBlock);
       AReport.Out_PosAsAttrs(CurPos);
       AReport.Out_XAttrEnd;
       {$ENDIF}
       {----}
       MarkTermsInTTColl;
       {----}
       g.LastEntry := nil; g.GenCount := 0; g.DemBase := g.VarNbr;
```

```
DefinitionList.Init(2,4);
while InFile.Current.Kind <> ikMscEndBlock do
begin gRedef := false;
fillchar(gCorrCond[1],SizeOf(gCorrCond)-SizeOf(pointer),0);
ItTyp := nil;
for nk := Low(NotationKind) to High(NotationKind) do
   lNotatExtCount[nk] := Notat[nk].fExtCount;
{ Inicjalizacje do obslugi konstruktorow i definiensu}
{ Initializations for handling constructors and definiens}
gWhichOne := 0;
gSuperfluous := 0; { dla definicji zostaje 0, dla redefinicji jest wyliczana }
                   { for definition it remains 0, for redefinition it is calculated }
gPropertiesOcc := false;
gDefNode.MeansOccurs := ' ';
case InFile.Current.Kind of
   ikItmGeneralization:
      begin
         lDeclBase := g.VarNbr;
         gExportableItem := true;
         gConstInExportableItemOcc := false;
         Parametrization;
         gExportableItem := false;
         gConstInExportableItemOcc := false;
         SkelList('D',lDeclBase);
      end;
   { Przy przyjeciu restrykcyjnej koncepcji dla typow lokusow nie ma sensu
     uzywac tej samej procedury dla generalizacji i parametryzacji.
   }
   { When adopting a restrictive concept for locus types, it does not make sense
     to use the same procedure for generalization and parameterization. }
   ikItmAssumption:
      begin
         gNonPermissive := false;
         InFile.InPos(lPos);
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart0(elAssume);
         {$ENDIF}
         InFile.InWord;
         gExportableItem := true;
         gConstInExportableItemOcc := false;
         ReadPropositions(lConditions);
         gExportableItem := false;
         gConstInExportableItemOcc := false;
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_Propositions(lConditions);
         AReport.Out_XElEnd(elAssume);
         {$ENDIF}
         InFile.InWord;
         lFrm := ConjugatePropositions(lConditions);
         SkelSnt('A',lFrm);
         lConditions.Done;
      end;
   ikItmExAssumption:
      begin
         gNonPermissive := false;
         lVarBase := g.VarNbr;
         InFile.InPos(CurPos);
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elGiven);
         AReport.Out_XIntAttr(atNr, g.VarNbr+1);
```

```
            AReport.Out_XAttrEnd;
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            GetConstQualifiedList;
            ReadPropositions(lConditions);
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            InFile.InWord;
            lFrm := xFormula(ConjugatePropositions(lConditions));
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_Propos(0, 0, CurPos, lFrm);
            {$ENDIF}
            SkelSnt('A',lFrm);
            WriteQualified;
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_Propositions(lConditions);
            AReport.Out_XElEnd(elGiven);
            {$ENDIF}
            lConditions.Done;
            for i := lVarBase+1 to g.VarNbr do FixedVar[i].nSkelConstNr := 0;
         end;
      ikItmDefMode:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'M');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            DefModePattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            Correctness;
            //   Uwaga obrobka blednych properties, na razie nie ma properties dla Modow
            { Please note: processing of incorrect properties, there are no properties for Mods yet }
            ProcessProperties('M');
            InsertMode;
            DefModeTail;
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmRedefMode:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'M');
            AReport.Out_XAttr(atRedefinition, 'true');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            RedefModePattern;
            gExportableItem := false;
```

```
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            Correctness;
            //   Uwaga obrobka blednych properties, na razie nie ma properties dla Modow
            { Please note: processing of incorrect properties, there are no properties for Mods yet }
            ProcessProperties('M');
            if (gSuperFluous <> 0) or gSpecified then InsertMode;
            DefModeTail;
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmDefExpandMode:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'M');
            AReport.Out_XAttr(atExpandable, 'true');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            DefExpandableMode;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            // ##TODO: this Correctness seems useless
            // wydaje sie potrzebna na potrzeby obslugi blednych sytuacji
            { seems to be needed to handle error situations }
            Correctness;
            //   Uwaga obrobka blednych properties, na razie nie ma properties dla Modow
            { Please note: processing of incorrect properties, there are no properties for Mods yet }
            ProcessProperties('M');
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmDefPrAttr:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'V');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            DefPredAttributePattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
```

```
            Correctness;
            //   Uwaga obrobka blednych properties, na razie nie ma properties dla Atrybutow
            { Please note: processing of incorrect properties, there are no properties for Attributes yet }
            ProcessProperties('V');
            InsertPredAttribute;
            DefAttrTail;
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmRedefPrAttr:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'V');
            AReport.Out_XAttr(atRedefinition, 'true');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            RedefPredAttributePattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            Correctness;
            //   Uwaga obrobka blednych properties, na razie nie ma properties dla Atrybutow
            { Please note: processing of incorrect properties, there are no properties for Attributes yet }
            ProcessProperties('V');
            if gSuperFluous <> 0 then
               InsertPredAttribute;
            DefAttrTail;
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmDefPred:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'R');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            DefPredPattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            Correctness;
            ProcessProperties('R');
            InsertPredicate;
            DefPredTail;
```

```
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmDefFunc:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'K');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            DefFuncPattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            Correctness;
            ProcessProperties('K');
            InsertFunctor;
            DefFuncTail;
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmRedefPred:
         begin
            {$IFDEF ANALYZER_REPORT}
            AReport.Out_XElStart(elDefinition);
            AReport.Out_XAttr(atKind, 'R');
            AReport.Out_XAttr(atRedefinition, 'true');
            {$ENDIF}
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            RedefPredPattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            {$IFDEF ANALYZER_REPORT}
            WriteDefiniensLabel;
            AReport.Out_XAttrEnd;
            {$ENDIF}
            Correctness;
            ProcessProperties('R');
            if (gSuperFluous <> 0) or gPropertiesOcc then InsertPredicate;
            DefPredTail;
            WriteDefiniens;
            {$IFDEF ANALYZER_REPORT}
            Do_Patterns;
            AReport.Out_XElEnd(elDefinition);
            {$ENDIF}
         end;
      ikItmRedefFunc:
         begin
            {$IFDEF ANALYZER_REPORT}
```

```
      AReport.Out_XElStart(elDefinition);
      AReport.Out_XAttr(atKind, 'K');
      AReport.Out_XAttr(atRedefinition, 'true');
      {$ENDIF}
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      RedefFuncPattern;
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      {$IFDEF ANALYZER_REPORT}
      WriteDefiniensLabel;
      AReport.Out_XAttrEnd;
      {$ENDIF}
      Correctness;
      ProcessProperties('K');
      if (gSuperFluous <> 0) or gSpecified or gPropertiesOcc then InsertFunctor;
      DefFuncTail;
      WriteDefiniens;
      {$IFDEF ANALYZER_REPORT}
      Do_Patterns;
      AReport.Out_XElEnd(elDefinition);
      {$ENDIF}
   end;
ikItmDefStruct:
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart(elDefinition);
      AReport.Out_XAttr(atKind, 'G');
      AReport.Out_XAttrEnd;
      {$ENDIF}
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      DefStruct;
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      // No sense for structures
      //     WriteDefiniensLabel;
      // ##TODO: this Correctness seems useless, and makes praphan
      //         ugly, because we have to take special care of it there
      //   Uwaga obsluga blednych correctness
      { Note handling incorrect correctness }
      Correctness;
      //   Uwaga obrobka blednych properties, na razie nie ma properties dla Atrybutow
      ProcessProperties('G');
      {$IFDEF ANALYZER_REPORT}
      Do_Patterns;
      AReport.Out_XElEnd(elDefinition);
      {$ENDIF}
   end;
ikItmCanceled:
   begin
      InFile.InWord;
      case InFile.Current.Kind of
         ikDefTheoremCanceled:
            begin
               gDefNode.Kind := ':';
               DefinitionList.Insert(new(DefNodePtr, Init(' ',':',0,0,nil,nil)));
            end;
         ikTheoremCanceled:
            ErrImm(278);
```

```
                ikSchemeCanceled:
                   ErrImm(279);
             end;
             InFile.InWord;
          end
      else Statement;
      end;
      if ItTyp<> nil then dispose(ItTyp,Done);
      DisplayLine(CurPos.Line,ErrorNbr);
      end;
      InFile.InPos(CurPos); InFile.InWord;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_EndPos(CurPos);
      AReport.Out_XElEnd(elDefinitionBlock);
      {$ENDIF}
      CreateDefinientia;
      pVarNbr := g.VarNbr;
      DefinitionalTheorems;
      g.VarNbr := pVarNbr;
      while g.LastEntry <> nil do
      begin
         lEntry := g.LastEntry^.PreviousEntry;
         with g.LastEntry^ do
            case FORM of
               'A','B': begin dispose(SkSnt,Done); dispose(dSnt,Done) end;
               'C','D': begin SkList.Done; SkIdents.Done; SkOrigTyps.Done; end;
            end;
         dispose(g.LastEntry);
         g.LastEntry := lEntry;
      end;
      DisposeTrmList(gDefiniendumArgs); gDefiniendumArgs := nil;
      CloseDef;
      DisposeLevel(d);
   end;
```

This code is used in chunk 114a.
Defines:
Definition, used in chunk 138.
Uses CloseDef 12b, ConjugatePropositions 9b, Correctness 24, CreateDefinientia 117c, D 12a, DefAttrTail 37c, DefExpandableMode 47b, DefFuncPattern 38b, DefFuncTail 43b, DefinitionalTheorems 119, DefinitionList 29a, DefModePattern 43d, DefModeTail 62b, DefNodePtr 4b, DefPredAttributePattern 47c, DefPredPattern 33b, DefPredTail 37b, DefStruct 64, DisposeLevel 78b, gDefiniendumArgs 16b, gDefNode 17a, GetConstQualifiedList 14a, InsertFunctor 43a, InsertMode 47a, InsertPredAttribute 50, InsertPredicate 37a, OpenDef 12a, Parametrization 114b, ProcessProperties 106b, ReadPropositions 9a, RedefFuncPattern 39b, RedefModePattern 44, RedefPredAttributePattern 48, RedefPredPattern 34c, SkelList 87b, SkelSnt 88a, Statement 100, WriteDefiniens 29b, WriteDefiniensLabel 17a, WriteQualified 14b, and xFormula 13a.

128 ⟨*Round up item* 128⟩≡

```
   // ##TODO: this very much resembles RoundUpTrmType, try to avoid
   //         such copying of code.
   // ##TODO: why do we use even the clusters from Count to fExtCount-1 here???
   //         It seems fairly inconsistent with other usage of them in analizer.
   //         Insert all clusters immediatelly as in preparator,
   //         to get rid of the mess.
   procedure RoundUpItem(Item: TTPairPtr);
   var
      i,lLeft,lRight: integer;
      lKey: FClusterObj;
      lClusterPtr: AttrCollectionPtr;
   label Inconsistent;
   begin
      with Item^ do
      begin
```

```
      lClusterPtr := CopyCluster(nTyp^.UpperCluster);
      lKey.nClusterTerm := nTrm;
      if FunctorCluster.FindInterval(@lKey, lLeft, lRight) then
         for i := lLeft to lRight do
         begin
            RoundUpWith(FunctorCluster.AtIndex(i),nTrm,nTyp,lClusterPtr);
            { Powinno sie tutaj zglosic blad !}
            {You should report a bug here!}
            if not lClusterPtr^.fConsistent then goto Inconsistent;
         end;
      for i := FunctorCluster.Count to FunctorCluster.Count + FunctorCluster.fExtCount-1 do
      begin
         RoundUpWith(FunctorCluster.Items^[i],nTrm,nTyp,lClusterPtr);
         { Powinno sie tutaj zglosic blad !}
         {You should report a bug here!}
         if not lClusterPtr^.fConsistent then goto Inconsistent;
      end;
      Inconsistent:
         dispose(nTyp^.UpperCluster,Done);
      nTyp^.UpperCluster := lClusterPtr;
   end;
end;
```

This code is used in chunk 114a.
Defines:
RoundUpItem, used in chunk 129.

129 ⟨*Parse a registration* 129⟩≡

```
procedure Registration;
var
   lDeclBase,i,z,pVarNbr: integer;
   lRoundUpClusters: boolean;
begin
   InFile.InPos(CurPos);
   gMaxArgNbr := 2*MaxArgNbr;
   gDefiniendumArgs := nil;
   d := g;
   D.LocPredNbr := LocPredDef.Count;
   D.LocFuncNbr := LocFuncDef.Count;
   OpenDef;
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_XElStart(elRegistrationBlock);
   AReport.Out_PosAsAttrs(CurPos);
   AReport.Out_XAttrEnd;
   {$ENDIF}
   {----}
   MarkTermsInTTColl;
   {----}
   g.GenCount := 0;
   g.DemBase := g.VarNbr;
   DefinitionList.Init(2,4);
   while InFile.Current.Kind <> ikMscEndBlock do
   begin
      fillchar(gCorrCond[1],SizeOf(gCorrCond)-SizeOf(pointer),0);
      ItTyp := nil;
      { Inicjalizacje do obslugi konstruktorow i definiensu}
      { Initializations for handling constructors and definiens}
      case InFile.Current.Kind of
         ikItmGeneralization:
            begin
               lDeclBase := g.VarNbr;
```

```
                gExportableItem := true;
                gConstInExportableItemOcc := false;
                Parametrization;
                gExportableItem := false;
                gConstInExportableItemOcc := false;
             end;
  { Przy przyjeciu restrykcyjnej koncepcji dla typow lokusow nie ma sensu
    uzywac tej samej procedury dla generalizacji i parametryzacji.
  }
  { When adopting a restrictive concept for locus types, it does not make sense
    to use the same procedure for generalization and parameterization.
  }
          ikItmCluRegistered:
             begin
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElStart0(elRegistration);
                {$ENDIF}
                gExportableItem := true;
                gConstInExportableItemOcc := false;
                DefExistentialCluster;
                gExportableItem := false;
                gConstInExportableItemOcc := false;
                Correctness;
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElEnd(elRegistration);
                {$ENDIF}
             end;
          ikItmCluConditional:
             begin
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElStart0(elRegistration);
                {$ENDIF}
                gExportableItem := true;
                gConstInExportableItemOcc := false;
                DefConditionalCluster;
                gExportableItem := false;
                gConstInExportableItemOcc := false;
                Correctness;
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElEnd(elRegistration);
                {$ENDIF}
             end;
          ikItmCluFunctor:
             begin
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElStart0(elRegistration);
                {$ENDIF}
                gExportableItem := true;
                gConstInExportableItemOcc := false;
                DefFunctorCluster;
                gExportableItem := false;
                gConstInExportableItemOcc := false;
                Correctness;
                {$IFDEF ANALYZER_REPORT}
                AReport.Out_XElEnd(elRegistration);
                {$ENDIF}
                { R e t r o s p e k t y w n e   z a o k r a g l a n i e   t y p o w }
                with gTermCollection do
                   for z := 0 to Count-1 do RoundUpItem(TTPairPtr(Items^[z]));
             end;
```

```
// ###TODO: why canceled clusters??? This may probably cause BUGS!
ikIdFunctors:
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elIdentifyRegistration);
      {$ENDIF}
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      DefIdentify(ikTrmFunctor);
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      Correctness;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elIdentifyRegistration);
      {$ENDIF}
   end;
ikIdPredicates:
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elIdentifyRegistration);
      {$ENDIF}
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      DefIdentify(ikFrmPred);
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      Correctness;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elIdentifyRegistration);
      {$ENDIF}
   end;
ikIdAttributes:
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elIdentifyRegistration);
      {$ENDIF}
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      DefIdentify(ikFrmAttr);
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      Correctness;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elIdentifyRegistration);
      {$ENDIF}
   end;
ikReduceFunctors:
   begin
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElStart0(elReductionRegistration);
      {$ENDIF}
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      DefReduction;
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      Correctness;
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elReductionRegistration);
      {$ENDIF}
```

```
            end;
         ikProperty:
            begin
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XElStart0(elPropertyRegistration);
               {$ENDIF}
               gExportableItem := true;
               gConstInExportableItemOcc := false;
               DefProperty;
               gExportableItem := false;
               gConstInExportableItemOcc := false;
               Justify(0,0,gPropertyCond);
               {$IFDEF ANALYZER_REPORT}
               AReport.Out_XElEnd(elPropertyRegistration);
               {$ENDIF}
            end
      else Statement;
      end;
      if ItTyp<> nil then dispose(ItTyp,Done);
      DisplayLine(CurPos.Line,ErrorNbr);
   end;
   InFile.InPos(CurPos); InFile.InWord;
   {$IFDEF ANALYZER_REPORT}
   AReport.Out_EndPos(CurPos);
   AReport.Out_XElEnd(elRegistrationBlock);
   {$ENDIF}
   lRoundUpClusters := (ConditionalCluster.fExtCount > 0)
      or (FunctorCluster.fExtCount > 0);
   DisposeTrmList(gDefiniendumArgs); gDefiniendumArgs := nil;
   CloseDef;
   DisposeLevel(d);
   gMaxArgNbr := MaxArgNbr;
   if lRoundUpClusters then
   begin
      NonZeroTyp^.RoundUp;
      for i := 0 to RegisteredCluster.Count-1 do
         with RClusterPtr(RegisteredCluster.Items^[i])^ do
      begin
         move(nPrimaryList.Items^,LocArgTyp[1],nPrimaryList.Count*sizeof(pointer));
         nConsequent.Upper^.RoundUpWith(nClusterType);
         gTermCollection.FreeAll;
      end;
   end;
   RemoveTermsFromTTColl;
end;
```

This code is used in chunk 114a.
Defines:
Registration, used in chunk 138.
Uses CloseDef 12b, Correctness 24, D 12a, DefConditionalCluster 53, DefExistentialCluster 52, DefFunctorCluster 54, DefIdentify 59, DefinitionList 29a, DefProperty 62a, DefReduction 58, DisposeLevel 78b, gDefiniendumArgs 16b, gMaxArgNbr 8b, gPropertyCond 62a, Justify 96b, OpenDef 12a, Parametrization 114b, RoundUpItem 128, and Statement 100.

132 ⟨*Parse notation* 132⟩≡

```
procedure Notation;
var
   lDeclBase,pVarNbr: integer;
   nk: NotationKind;
begin
   InFile.InPos(CurPos);
   gDefiniendumArgs := nil;
   d := g;
```

```
D.LocPredNbr := LocPredDef.Count;
D.LocFuncNbr := LocFuncDef.Count;
OpenDef;
{$IFDEF ANALYZER_REPORT}
AReport.Out_XElStart(elNotationBlock);
AReport.Out_PosAsAttrs(CurPos);
AReport.Out_XAttrEnd;
{$ENDIF}
{----}
MarkTermsInTTColl;
{----}
g.LastEntry := nil;
g.GenCount := 0;
g.DemBase := g.VarNbr;
DefinitionList.Init(2,4);
while InFile.Current.Kind <> ikMscEndBlock do
begin
   ItTyp := nil;
   nk := noForgetFunctor; // used as the uninitialised value for patterns here
   { Inicjalizacje do obslugi konstruktorow i definiensu}
   { Initializations for handling constructors and definiens}
   gWhichOne := 0;
   gSuperfluous := 0; { dla definicji zostaje 0, dla redefinicji jest wyliczana }
                      { for definition it remains 0, for redefinition it is calculated }
   case InFile.Current.Kind of
      ikItmGeneralization:
         begin lDeclBase := g.VarNbr;
         gExportableItem := true;
         gConstInExportableItemOcc := false;
         Parametrization;
         gExportableItem := false;
         gConstInExportableItemOcc := false;
         end;
      { Przy przyjeciu restrykcyjnej koncepcji dla typow lokusow nie ma sensu
        uzywac tej samej procedury dla generalizacji i parametryzacji.
      }
      { When adopting a restrictive concept for locus types, it does not make sense
        to use the same procedure for generalization and parameterization. }
      ikItmDefMode:
         begin
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            NotatModePattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            nk := noMode;
         end;
      ikItmDefPred:
         begin
            gExportableItem := true;
            gConstInExportableItemOcc := false;
            NotatPredPattern;
            gExportableItem := false;
            gConstInExportableItemOcc := false;
            nk := noPredicate;
         end;
      // ###TODO: ikItmDefAttr and ikItmRedefAttr are no longer used in anal,
      //           it may be a dead code in parser too - fix it
      ikItmDefPrAttr:
         begin
```

```
                gExportableItem := true;
                gConstInExportableItemOcc := false;
                NotatPredAttributePattern;
                gExportableItem := false;
                gConstInExportableItemOcc := false;
                nk := noAttribute;
             end;
          ikItmDefFunc:
             begin
                gExportableItem := true;
                gConstInExportableItemOcc := false;
                NotatFuncPattern;
                gExportableItem := false;
                gConstInExportableItemOcc := false;
                nk := noFunctor;
             end;
          // ten Staement jest dla celow Errors Recovery na przypadek bledow syntaktycznych
          { This Statement is for Errors Recovery purposes in case of syntactic errors }
       else Statement;
       end;
       if ItTyp<> nil then dispose(ItTyp,Done);
       DisplayLine(CurPos.Line,ErrorNbr);
       {$IFDEF ANALYZER_REPORT}
       if nk <> noForgetFunctor then
          with Notat[nk] do
             AReport.Out_Pattern(Items^[Count + fExtCount - 1],
                                 Count + fExtCount);
       {$ENDIF}
    end;
    InFile.InPos(CurPos); InFile.InWord;
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_EndPos(CurPos);
    AReport.Out_XElEnd(elNotationBlock);
    {$ENDIF}
    DisposeTrmList(gDefiniendumArgs); gDefiniendumArgs := nil;
    CloseDef;
    DisposeLevel(d);
 end;
```

This code is used in chunk 114a.
Defines:
Notation, used in chunk 138.
Uses CloseDef 12b, D 12a, DefinitionList 29a, DisposeLevel 78b, gDefiniendumArgs 16b, NotatFuncPattern 41, NotatModePattern 45, NotatPredAttributePattern 49, NotatPredPattern 35, OpenDef 12a, Parametrization 114b, and Statement 100.

134 ⟨*Parse a scheme block* 134⟩≡

```
procedure Scheme;
var
   kk: integer;
begin
   InFile.InPos(CurPos);
   {----}
   MarkTermsInTTColl;
   {----}
   gExportableItem := true;
   gConstInExportableItemOcc := false;
   SchemeBody;
   gExportableItem := false;
   gConstInExportableItemOcc := false;
   Infile.InWord;
   Demonstration(0,0,gSchemeThesis);
```

```
    dispose(gSchemeThesis,Done);
    CurSchFuncTyp.Done;
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_EndPos(CurPos);
    AReport.Out_XElEnd(elSchemeBlock);
    {$ENDIF}
    for kk := 1 to CurSchFuncNbr do SchFuncArity[kk].nArity.Done;
    for kk := 1 to gSchPredNbr do SchPredArity[kk].nArity.Done;
    RemoveTermsFromTTColl;
  end;
```

This code is used in chunk 114a.
Defines:
Scheme, used in chunk 138.
Uses CurSchFuncNbr 9c, Demonstration 85, gSchemeThesis 9c, gSchPredNbr 9c, and SchemeBody 9c.

135a ⟨*Analyze reduction-like theorem* 135a⟩≡

```
  {$IFDEF THEOREM2REDUCTION}
  var fileTh2Red: text;
  const fileTh2RedName = 'th2red.txt';
  var ThNr: word;

  function ReductionLikeTheorem(f: FrmPtr): boolean;
  var
     lPredNr: integer;
     lArgs: TrmList;
  begin
     case f^.FrmSort of
        ikFrmPred:
           begin
              AdjustFrm(PredFrmPtr(f),lPredNr,lArgs);
              if lPredNr = gBuiltIn[rqEqualsTo] then
                 ReductionLikeTheorem :=
                    ReductionAllowed(lArgs^.XTrmPtr,lArgs^.NextTrm^.XTrmPtr) or
                    ReductionAllowed(lArgs^.NextTrm^.XTrmPtr,lArgs^.XTrmPtr)
              else ReductionLikeTheorem := false;
           end;
        ikFrmUniv: ReductionLikeTheorem := ReductionLikeTheorem(UnivFrmPtr(f)^.Scope);
        //  ikFrmNeg: ReductionLikeTheorem := ReductionLikeTheorem(NegFrmPtr(f)^.NegArg);
     else ReductionLikeTheorem := false;
     end;
  end;
  {$ENDIF}
```

This code is used in chunk 114a.
Defines:
fileTh2Red, used in chunks 135b and 138.
ReductionLikeTheorem, used in chunk 135b.
ThNr, used in chunks 135b and 138.

135b ⟨*Parse a theorem* 135b⟩≡

```
  procedure Theorem;
  var
     lFrm: FrmPtr;
     lLabNr,lLabId: integer;
  begin
     InFile.InPos(CurPos);
     {$IFDEF ANALYZER_REPORT}
     AReport.Out_XElStart(elJustifiedTheorem);
     AReport.Out_PosAsAttrs(CurPos);
     AReport.Out_XAttrEnd;
     {$ENDIF}
     InFile.InWord;
```

```
      lLabNr := InFile.Current.Nr;
      InFile.InInt(lLabId);
      InFile.InPos(CurPos);
      InFile.InWord;
      gExportableItem := true;
      gConstInExportableItemOcc := false;
      lFrm := ReadSentence(false);
      {$IFDEF THEOREM2REDUCTION}
      if ReductionLikeTheorem(lFrm) then
      begin
         ErrImm(701);
         if ThNr = 1 then writeln(fileTh2Red,MizFileName);
         writeln(fileTh2Red,CurPos.Line, ' ',CurPos.Col, ' ', 701);
         inc(ThNr);
      end;
      {$ENDIF}
      gExportableItem := false;
      gConstInExportableItemOcc := false;
      Justify(lLabNr,lLabId,lFrm);
      dispose(lFrm,Done);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XElEnd(elJustifiedTheorem);
      {$ENDIF}
   end;
```

This code is used in chunk 114a.
Defines:
Theorem, used in chunk 138.
Uses fileTh2Red 135a, Justify 96b, ReadSentence 7a, ReductionLikeTheorem 135a, and ThNr 135a.

136a ⟨*Parse section* 136a⟩≡

```
   procedure Section;
   begin
      InFile.InPos(CurPos);
      {$IFDEF ANALYZER_REPORT}
      AReport.Out_XEl1(elSection);
      {$ENDIF}
      InFile.InWord;
   end;
```

This code is used in chunk 114a.
Defines:
Section, used in chunk 138.

136b ⟨*Parse cancelled item* 136b⟩≡

```
   procedure Canceled;
   var
      lThProp: PropositionPtr;
   begin
      InFile.InWord;
      if InFile.Current.Kind = ikTheoremCanceled then
      begin
         lThProp := new(PropositionPtr, Init(0, 0, NewVerum, CurPos));
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elJustifiedTheorem);
         AReport.Out_PosAsAttrs(CurPos);
         AReport.Out_XAttrEnd;
         AReport.Out_Proposition(lThProp);
         AReport.Out_XEl1(elSkippedProof);
         AReport.Out_XElEnd(elJustifiedTheorem);
         {$ENDIF}
         dispose(lThProp, Done);
      end
```

```
      else
      begin
         {$IFDEF ANALYZER_REPORT}
         AReport.Out_XElStart(elCanceled);
         AReport.Out_XAttr(atKind, InFile.Current.Kind);
         AReport.Out_XElEnd0;
         {$ENDIF}
      end;
      InFile.InWord;
   end;
```

This code is used in chunk 114a.
Defines:
Canceled, used in chunk 138.

137a ⟨*Load SGN environment file(?)* 137a⟩≡

```
   procedure LoadSGN;
   var
      Antonym: boolean;
      lPattern: PatternPtr;
      lInEnvFile: InEnvFilePtr;
      nk: NotationKind;
   begin
      FileExam(EnvFileName+'.eno');
      lInEnvFile := new(InEnvFilePtr,OpenFile(EnvFileName+'.eno'));
      with lInEnvFile^ do
      begin
         NextElementState;
         XMLASSERT(nElKind = elNotations);
         NextElementState;
         for nk := Low(NotationKind) to High(NotationKind) do
            Notat[ nk].Init(MaxNotatNbr(nk));
         while not (nState = eEnd) do
         begin
            XMLASSERT(nElKind = elPattern);
            lPattern := In_Pattern;
            Notat[lPattern^.fKind].Insert(lPattern);
            gTermCollection.FreeAll;
         end;
      end;
      dispose(lInEnvFile,Done);
      for nk := Low(NotationKind) to High(NotationKind) do
         NotatBase[nk] := Notat[nk].Count;
   end;
```

This code is used in chunk 114a.
Defines:
LoadSGN, used in chunk 138.

137b ⟨*Dispose analyze* 137b⟩≡

```
   procedure DisposeAnalyze;
   var
      nk: NotationKind;
      gg: LevelRec;
   begin
      Definientia.Done;
      gIdentifications.Done;
      gReductions.Done;
      gPropertiesList.Done;
      for nk := Low(NotationKind) to High(NotationKind) do
         Notat[nk].Done;
      DisposeConstructors;
      dispose(AnyTyp,Done);
```

```
    with gg do
    begin
       VarNbr := 0;
       LocPredNbr := 0;
       LocFuncNbr := 0;
    end;
    DisposeLevel(gg);
    gTermCollection.Done;
    {-writeln(InfoFile,'Koniec analizatora, MemAvail=',MemAvail);
     InfoHeap;-}
  end;
```

This code is used in chunk 114a.
Defines:
DisposeAnalyze, used in chunk 3b.
Uses DisposeLevel 78b.

138 ⟨*Analyze* 138⟩≡

```
  procedure Analyze;
  var
     kk: integer;
     c: ConstructorsKind;
  begin
     {$IFDEF THEOREM2REDUCTION}
     Assign(fileTh2Red,fileTh2RedName);
     if MFileExists(fileTh2RedName) then Append(fileTh2Red) else Rewrite(fileTh2Red);
     ThNr := 1;
     {$ENDIF}
     {}
     Load_EnvConstructors;
     gAttrCollected := false;
     {}
     for c := Low(ConstructorsKind) to High(ConstructorsKind) do
        ConstrBase[c]      := Constr[c].Count;
     RegClusterBase := RegisteredCluster.Count;
     FuncClusterBase := FunctorCluster.Count;
     CondClusterBase := ConditionalCluster.Count;
     gDefNode.fPrimaries.Init(0,1);
     AnyTyp := new(TypPtr,Init(ikTypMode,NewEmptyCluster,NewEmptyCluster,gBuiltIn[rqAny],Nil));
     ResNbr := 0;
     with g do begin VarNbr := 0; LocPredNbr := 0; LocFuncNbr := 0 end;
     {$IFDEF ANALYZER_REPORT}
     AReport.OpenFileWithXSL(MizFileName+'.xml');
     AReport.Out_XElStart(elArticle);
     AReport.Out_XAttr(atAid, ArticleID);
     AReport.Out_XMizQuotedAttr(atMizfiles, MizFiles);
     AReport.Out_XAttrEnd;
     {$ENDIF}
     LoadSGN;
     { obsluga nieoczekiwanych warunkow }
     { handling unexpected conditions }
     gCorrCond[0] := NewIncorFrm;
     Definientia.Init(20);
     if Verifying then LoadDefinitions;
     gIdentifications.Init(0);
     gReductions.Init(0);
     gPropertiesList.Init(0);
     LoadPropertiesReg;
     RegPropertiesBase := gPropertiesList.Count;
     InFile.OpenFile(MizFileName+'.par');
     InFile.InWord;
     {$IFDEF ANALYZER_REPORT}
```

```
    DoCtrans := false; DoStrans := false;
    {$ENDIF}
    while InFile.Current.Kind<>'!' do
    begin
       case InFile.Current.Kind of
          ikBlcSection: Section;
          ikBlcDefinition: Definition;
          ikBlcRegistration: Registration;
          ikBlcNotation: Notation;
          ikItmReservation: Reservation;
          ikBlcScheme: Scheme;
          ikItmTheorem: Theorem;
          ikItmCanceled: Canceled;
       else Statement;
       end;
       {$IFDEF ANALYZER_REPORT}
       AReport.OutNewLine;
       {$ENDIF}
       DisplayLine(CurPos.Line,ErrorNbr);
    end;
    Infile.Done;
    {$IFDEF ANALYZER_REPORT}
    AReport.Out_XElEnd(elArticle);
    AReport.Done;
    {$ENDIF}
    dispose(gCorrCond[0],Done);
    for kk := 1 to ResNbr do dispose(ReservedVar[kk],Done);
    {$IFDEF THEOREM2REDUCTION}
    Close(fileTh2Red);
    {$ENDIF}
  end;
```

This code is used in chunk 114a.
Defines:
  Analyze, used in chunks 3b, 7, 8, 27a, 52–54, 58, 70, and 100.
Uses Canceled 136b, Definition 121, fileTh2Red 135a, gDefNode 17a, LoadSGN 137a, Notation 132, Registration 129, Reservation 70, Scheme 134, Section 136a, Statement 100, Theorem 135b, and ThNr 135a.