# Internship at the Naproche Project from April 2009 to July 2009

John W. Schmid

Mathematical Institute, University of Bonn
jwschmid@uni-bonn.de
http://www.naproche.net

## 1 The Naproche Project

What is a mathematical proof? How do mathematicians present proofs in scientific literature? And can a computer "read" and "calculate" the same proof as a mathematician? These are some of the core questions the Naproche (*Natural Language Proof Checking*) project tries to answer. It is a joint initiative of Peter Koepke at the Mathematical Institute of University of Bonn, Bernhard Schröder at the Linguistics Department of University Duisburg-Essen and Gregor Büchel at the Computer Science Department of the University of Applied Sciences of Cologne. Using the declarative programming language Prolog, the scientists at Naproche are constructing a system which lets textbook style proofs be checked for correctness by a computer. Step one in this is for the program to accept text formed from a controlled subset of ordinary mathematical language and to transform this text into formal statements. Step two is converting these formal representations into first-order logic so that an automated theorem prover (ATP) can check them. The "language" these formal representations are formulated in is a unique linguistic construct called *Proof Representation Structure* (PRS).

Where was my work area within Naproche? Figure 1 shows the architecture of the Naproche system. The boxes represent the major modules. Next to the downward arrows, the output format of the corresponding module is shown. The files mentioned explicitly in the boxes are the ones I worked with.

The procedure is the following: At first, the user submits his input; he may make use of the help file *help.html* and the three examples provided on naproche.net. By means of simple input processing, this input is transformed into sentences and words. If these sentence obeys the Naproche grammar, that is, if it is parsed by *dcg_error.pl*, a PRS, representing the semantics of the sentence, can be created. This is done using *dcg.pl*, the subprogram that constituted my main work area. Finally, *checker.pl* forms first-order formulae from the PRS; these can be checked by an ATP. During these steps, error and warning messages provide a feedback to the user. If the input can be proven, the output is "Theorem".

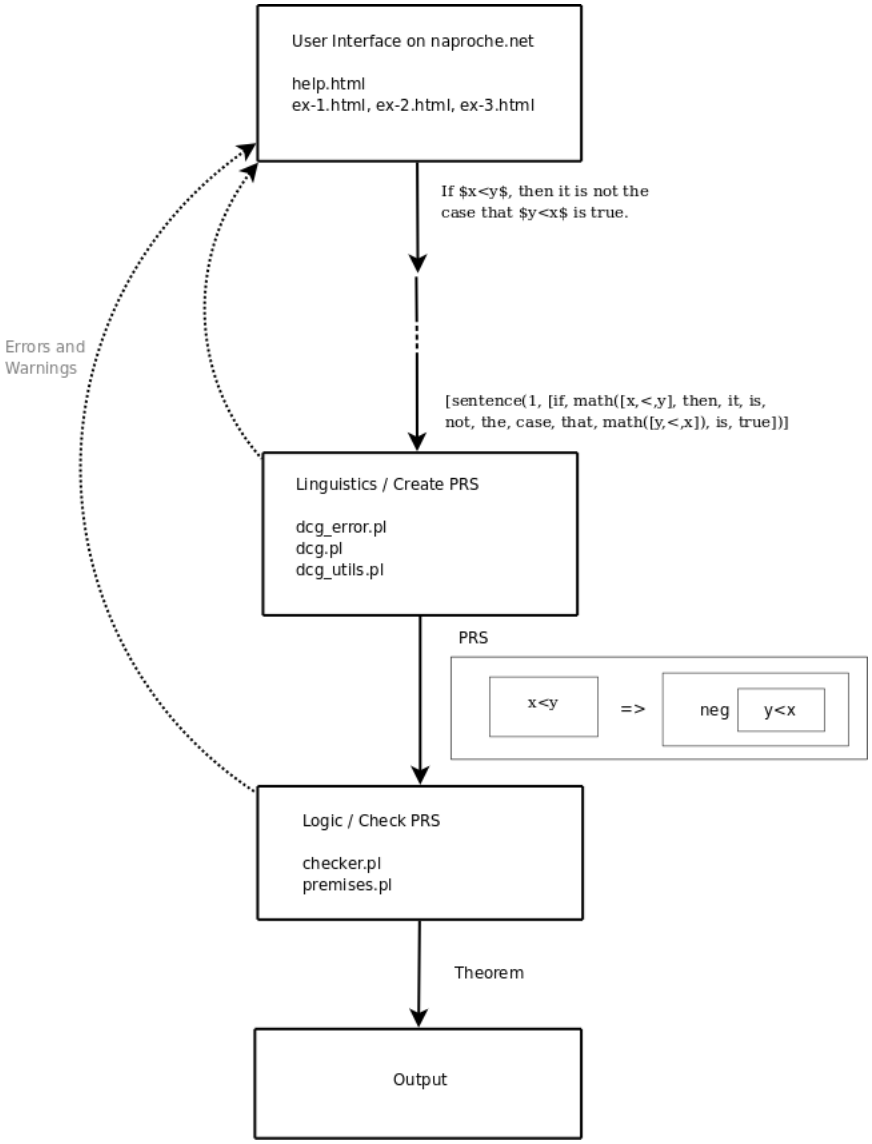


**Fig. 1.** The architecture of the Naproche system.

## 2 PRSs and Prolog

During the first one-and-a-half weeks, I dug into some of the scientific theory that contributes to the functioning of the Naproche programs. Namely, a relatively recent book in computer linguistics called "Representation and Inference for Natural Language. A First Course in Computational Semantics." by Blackburn and Bos. I solved many exercises of the chapters I read, even if homework was involved. They helped me to get a better grasp of the underlying principles. Their topic are *Discourse Representation Structures* (DRSs), a technique from Computational Linguistics designed to automatically extract the semantics of a controlled natural language text. The PRSs used in Naproche are a modification of these structures. I will give a brief explanation about what a PRS is shortly.

In my second week, I had a self-taught crash-course in Prolog, the programming language in which the Naproche program is written. I had to put in some effort to familiarize with - sit venia verbo - the *nature* of Prolog, i.e. using a declarative language instead of an imperative one. Contemplating the concept of the Naproche program, it becomes quite clear why declarativeness is advantageous: we give our computer sentences forming a textbook proof and ask him to check it for correctness. For this, we tell him exactly which formal statements need to be formed from the sentences. We tell him as if we were defining something, and not commanding.

In this period, I also learnt how to implement tests in Prolog, which is a useful tool for finding errors. For every new feature added to the Naproche program, at least one extra test is run regularly. This way, one makes sure that with every update the current functions of the program are conserved. During my internship, I wrote tests for every predicate I programmed or modified. Two further debugging methods are the Prolog predicates *write* and *trace*. Imagine *write* like a marker you put into the program code that creates extra output and tells you the program passed the point you marked. Meanwhile, *trace* allows you to follow the exact steps the Prolog processor made, one by one. I used the latter one especially often during my later assignments. Furthermore, I learnt how to write the documentation for my modifications in standard pldoc style. But now for the things I programmed.

## 3 The predicate "expecting_proof_and_qed"

My first real programming task was the following. The controlled mathematical language of the Naproche program requires proofs to follow a certain construction plan: a sentence saying "Theorem." or "Lemma.", a mathematical statement (called *goal*), then a sentence of the form "Proof.", mathematical statements (i.e. the proof itself), and finally the sentence "Qed.". This is in accordance with the standard style of presenting mathematical proofs found in basically all professional literature. Nevertheless, this is a restriction for the Naproche user, since other texts simply aren't recognized as proofs.

As for my task: I implemented a predicate that checked whether the input obeyed this construction plan or not. In other words, whenever a sentence saying

“Theorem.” or “Lemma.” occurred, the Prolog engine called my predicate expecting_proof_and_qed and succeeded only if the following input had the required form. The code presents a pattern of recursive definitions typical for Prolog: if the input sentence reads “Proof.”, go on with expecting_qed, otherwise, parse the next sentence, and so on and so forth. In case no such sentence saying “Proof.” is found, the Prolog engine fails to parse the input and produces an error message returning the sentence number and the message:
This theorem is missing a ”proof“.

After writing the predicate, I inserted it into *dcg.pl*, which handles the recognition of the grammar and builds PRSs from text. Later on, it was moved to the auxiliary document *dcg_utils*. It is now called by *dcg_error*, which acts as a preprocessor that checks the whole input for grammatical correctness before the semantics of the text are touched. This makes the compiler recognize errors much more quickly. It was a very adequate first task for me.

## 4 Metasentences, Part I

### 4.1 What is a PRS?

Unfortunately, I cannot quickly explain how Prolog works. But, before talking about metasentences, I will quickly allude the concept of a PRS. For more detailed information, see Kühlwein[1].
When thinking of a PRS, imagine a *box* containing the following five constituents:

- The Identification number (*id*)
- A list of mathematical referents (*mrefs*)
- A list of discourse referents (*drefs*)
- A list of textual referents (*rrefs*)
- An ordered list of conditions (*conds*).

The various referents are the subjects of the statements made in the Naproche program, i.e. they are the objects which (un-)quantified variables are linked to, whereas the conditions are statements themselves, possibly containing other statements. How is this done? Here are some important conditions a PRS can have:

- for any $n$-ary predicate $p$ and drefs $X_1$ through $X_n$, $predicate(X_1, \ldots, X_n, p)$
- $holds(X)$, representing the claim that the formula referenced by the dref $X$ is true.
- $math\_id(X, Y)$, which binds a discourse referent $X$ to a mathematical referent $Y$ (a formula or a term).
- $PrsA$
- $neg(PrsA)$, representing a negation.

[1] Daniel Kühlwein: A calculus for Proof Representation Structures, Bonn 2009, published on www.naproche.net

- $PrsA \Rightarrow PrsB$, representing an implication $A$ and the set of claims $B$ made inside the scope of this implication. If $PrsA$ contains no conditions, the condition represents a universally quantified formula
- $PrsA \vee PrsB$, representing a disjunction
- $>< [PrsA, PrsB, \ldots]$, representing an exclusive disjunction (a statement of the form "exactly one of the following PRS-conditions is true")
- NEW: $<> [PrsA, PrsB, \ldots]$, representing a statement of the form "at most one of the following PRS-conditions is true",

where PrsA and PrsB are PRSs themselves. Let's look at an easy example: see Fig. 2.

There are three PRSs within the big PRS we are looking at: two PRSs connected by a right-arrow implication that is the only condition in the big PRS, and one within the right one of these two[2]. In this case, we say the big PRS subordinates the two medium-level ones, and the right one subordinates the negated one. How does this matter? Well, did you notice that the variables x and y are introduced in the left PRS, in which they appear, but in the negated one, they appear without having been introduced in that own PRS's dref list. This is so, because the drefs of the lefthand PRS are *accessible* to the negated one. In the Naproche programming code, each PRS possesses discourse referents accessible *before* the PRS itself and discourse referents discourse referents accessible *after* it. The rules for accessibility follow an intuitive visual concept: PRSs that lie inside a given PRS or that have an arrow pointing from a given PRS to it, possess all the accessible variables that the given PRS does (and possibly more). Then again, a computer can hardly rely on its visual intuition, which is why we must feed him these rules when we tell him how to build PRSs, i.e. in the module *dcg.pl*.

### 4.2 How does a metasentence work?

So what is a metasentence? For Naproche, a metasentence is a sentence that makes statements about other sentences containing mathematical content. An everyday example, not from Naproche, would be

"1 + 1 = 2. This is true.".

The second sentence is a metasentence: it refers to the precedent sentence. What's more, to decide whether the second sentence is true or false, it is inevitable to know whether the first one is correct or not. So much for the very basic concept.

Now, the trouble with parsing *this* very example lies in the much-studied problem of pronoun resolution; here I will spare everyone from the details of this field. So what is a metasentence the Naproche program accepts?

[2] The right-hand-side of such an implication (or an assumption or likewise) is called a *scope*.

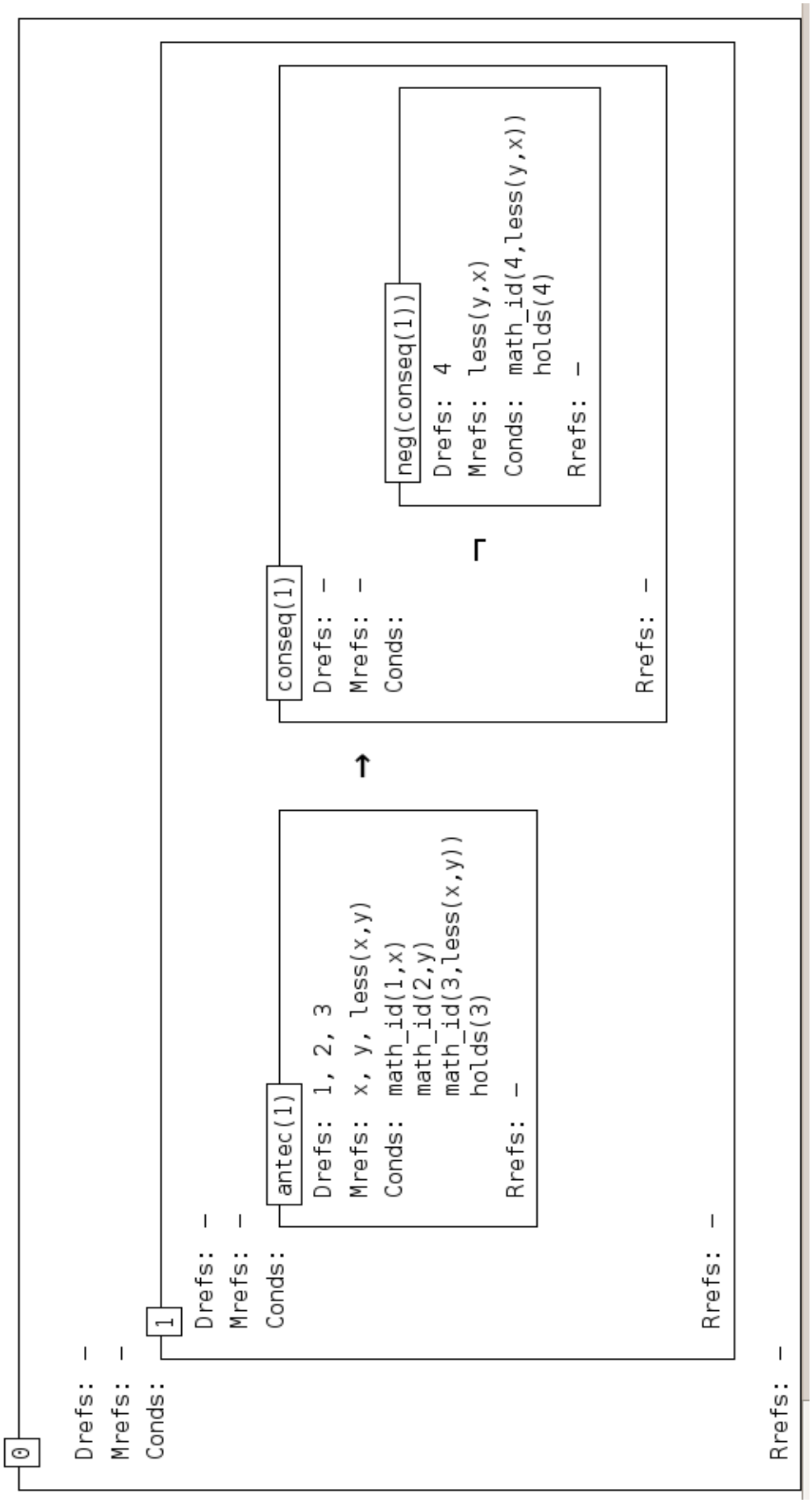


**Fig. 2.** An example of a PRS.

"Case one. $a < b$.[..]$Casetwo.b < a$. [..] Case one or case two is true."

The last sentence here is a metasentence in the meaning of Naproche. It refers to a case distinction made previously. Cases consist of an ID and a formal statement that characterizes them, called the case restrictor. The job of the predicate *metasentences* is to parse this kind of sentence. The sentence always consists of a noun phrase, here "Case one or case two", and a verb phrase, here "is true". Now, the tricky part of metasentences was the following. The information concerning the case restrictors is obviously contained in the noun phrases. But the information on how to logically connect the case restrictors is contained both in the noun phrase, i.e. "X or Y", and in the verb phrase, i.e. "is true", so we need to take PRS X and PRS Y and form a PRS containing the condition X $\vee$ Y. But in the moment when we parse the noun phrase, we do not know all this yet. Now look at the code for the predicate *metasentence*:

```
metasentence(MetaPRS,mode~Mode) -->
trigger(type~statement),
{
MetaPRS = id~Id..accbefore~AccBefore
},
meta_np(PRSList,Junctor,Id,AccBefore),
meta_vp(PRSList,Junctor,MetaPRS,mode~Mode).
```

The parameter just refers to the grammatical mode of the verb at the end of the sentence. *Mode* can be either *finite, infinitive*, or *to-infinitive*. PRSs are created from the cases listed in the noun phrase and put into a list of PRSs called PRSList. This parameter, PRSList, is constructed within the predicate *meta_np* and then used in meta_vp. Meanwhile, the parameter called "Junctor", crucial for the metasentence's formal statement, is determined within *meta_np* and *meta_vp*. In the example sentence, "X and Y are true", the np "X and Y" qualifies either as a meta_np with the junctor "and" or "neg_and" (negated and), while the vp "are true" qualifies as a meta_vp with the junctor "and" or with others, but not with the juntor "neg_and". This way, meta_np and meta_vp determine this parameter together. An extra difficulty in determining the junctor of a metasentence is contributed by commas. The words "Case one, case two," could either belong to an AND-statement or to an OR-statement, until one of the two connectors actually gets parsed. What Prolog does is to try out and parse one of them and either succeed or fail. Implementing this cost some effort.

Another important detail concerns the accessibility of variables. With all Junctors except "and", the variables of the PRS for case 1 are not accessible to the PRS for case 2. This is sensible, since we are dealing with two distinct cases, which need not both be true nor be connected with each other at all. If, for instance, case 1 says: "There is an a such that a¡b." and case 2 says: "a is odd.", then we don't want the Prolog engine to confuse the two distinct variables called a, unless case 1 and case 2 are both supposed true. Following the guidelines for PRS construction described above, we thus need to construct altogether different PRSs. In case the Junctor is "and", the drefs accessible after case 1 become the

ones accessible before case 2. In case it is not, the drefs accessible before case 2 are the same ones that are accessible before case 1.

As mentioned, the freshly built list of PRSs is then somehow logically connected within meta_vp. To do this, I implemented the following auxiliary predicates:

- *meta_merge* takes a list of PRS and merges them, using the existent predicate *prs_merge*.
- *negate_cases* takes a list of PRSes and makes a list of their negations. This list consists of conditions!
- *negate_cases_in_boxes* takes a list of PRSes and makes a list of PRSes containing their negations.
- *disjunct_cases* takes the PRS list and recursively connects PRSes through "or".

All in all, the predicate *metasentences* takes a certain sentence, extracts its case-restrictors and a logical connection and forms a PRS from these. We will see more examples of metasentences after the next two chapters.

## 5 Modifying Case Distinction

First, I will explain what I mean by a "case distinction". A case distinction is a text within a proof that contains cases in which different assumptions are made. A classical example is the following:

```
Proof. [...] Now we have three cases. Case one. x < y. Then f(x) = f(y).
Case two. x = y. In particular f(x) = f(y).
Case three. x > y. Hence f(x) = f(y).
So in all cases f(x) = f(y). Qed.
```

Here we have another one:

```
[...] Now precisely one of the following cases holds.
Case one. u < v. Case two. u > v.
So it is not the case that u = v. [...]
```

During the process of implementing the predicate *metasentence*, Marcos Cramer and I also modified the predicates *text, cases_text, exclusive_cases* and *cases*. This necessity arose because we found the following bug. If we submitted a proof containing a case distinction with a metasentence afterwards, the Naproche program did not just jump back to the "original" level of the proof, but treated the metasentence as a part of the last single case, instead. So how do the predicates work now?

### 5.1 The predicate "text"

The predicate *text* has a very fundamental function in *dcg.pl* and can be called at almost any point of input processing.Text contains statements, definitions,

assumptions, theorems and cases. There used to be a distinction between "text", where there necessarily to be a sentence or more, and "text_tail", which was space possibly filled by sentences. Also, there used to be a distinciton between text in- and outside theorems. What did no yet exist, though, was text within a certain case distinction. We made a single predicate from these types of text, adding two GULP-features to it named "type" and "empty". *Type* can assume the modi *normal, all_ass_closed, theorem* and *in_case. Empty* can assume the modi *no* and *possible*.

So how exactly is a case distinction possible? In the predicate *text*, case distinctions are divided into two types.

– During any text, we may introduce cases via *cases_text*, close them with a closing statement (see below), and continue with text.

```
text(In,Out,type~Type) -->
cases_text(In,TmpOut,type~Type),
text(TmpOut,Out,type~Type..empty~possible).
```

– At the end of a proof, however, if we aren't in a case distinction already, we may introduce cases and don't need to close them.

```
text(In,Out,type~Type) -->
{
\+ Type = in_case
},
cases(In,TmpOut,[],CaseList,type~Type..subtype~beginning),
{
% this predicate adds a condition stating that
% (at least) one of the cases that have been introduced must hold.
add_case_distinction(CaseList,TmpOut,Out)
}.
```

Here, *add_case_distinction* takes an ingoing PRS and a list of cases, disjuncts the cases and adds the resulting PRS to the original PRS's conditions (confer code). It makes use of the auxiliary predicate *disjunct_cases* described earlier; both predicates are implemented in *dcg_utils.pl*.

### 5.2 The predicate "cases_text"

*Cases_text* can either be used for listing mutually exclusive cases, or for proving something by case distinction and finishing it with a closing statement. The important point is that, either way, a complete "text about case distinction" is there, as opposed to the predicate *cases*, which only handles a list of cases.

– Mutually exclusive cases are introduced either by the sentence

```
Precisely one of the following cases holds.
```

or by

```
At most one of the following cases holds.
```

and the rest of the input is processed by *exclusive_cases*. Note that in this version, only the case assumptions (i.e. case-restrictors) are listed, no statements connected with them!

– For proving a simple (non-exclusive) case distinction, the cases and their subsequent texts are first called by *cases*, producing case-restrictors and scopes (see: What is a PRS?). Then, *add_case_distinction* is applied to the list of case-restrictors. This ensures that one of the cases really holds. And finally, a last sentence in a specific format needs to be parsed: it must contain the words "in all cases" before the noun. This way, we know that the case distinction is over and we jump back to the higher semantical level (before the case assumptions like 'If x<y, ...').

### 5.3 The predicate "exclusive_cases"

This predicate consists of a mere list of cases that are to be inserted in a "><-condition" (if exactly one case is true) or a "<>-condition" (if at most one case is true).
And that is precisely what it does: it works its way through the PRS list and forms the respective condition, adding each PRS to its objects (see example). To translate ><- or <>-condition into more explicit logical statements is left to the checker![3]

### 5.4 The predicate "cases"

*Cases* processes a list of single cases and proofs attached to each of them. A single case contains the sentence "case n: Statement" and then Text. Statement and Text are both transformed into PRSes. An implication condition Statement ==> Text is added to the outgoing PRS, and Statement is added to the list of case-restrictors. Before the first case, an introductory sentence is possibe confer *case_introduction*.
Besides that, the predicate *cases* possesses the GULP-feature "Type", same as *text* and *cases_text* do. Like these predicates, in *cases*, *type* can be *in_case*, indicating that the text is situated within a case distinction. Case distinctions within case distinctions are possible, here *cases_text* is called by *text*; confer the appended example.
But what's more, *cases* features a "Subtype", indicating whether we are dealing with the first case or not. This implementation has to do with the introductory sentence contained by *case_introduction*. *Case_introduction* contains either a sentence saying "there are N cases" (N>=2) or nothing. If SubType is 'in_case', this sentence is obligatory. On the other hand, if we are just presenting the second case, the sentence is not permitted.

[3] This sort of implementation, that adds a symbol to the controlled natural language which is logically redundant but appealing, is called "syntactic sugar".

## 6 Metasentences, Part II

Now for the promise I made earlier: I will enumerate the different types of statements in metasentences, characterized by the Junctors. All of this happens within the realms of the predicate *meta_vp*. remember the words in meta_vp are either an affirmative or a negative clause. meta_vp takes the list of case-PRSs from meta_np and does one of eight things:

- 1. affirmative, and: conjunct them.
- 2. negative, and: negate, then conjuct them.
- 3. affirmative, or: disjunct them.
- 4. negative, or: negate, then disjunct them.
- 5. 'precisely one case holds': puts the list in a '><' condition.
- 6. 'precisely one case is false': negates them, puts them in a '><' condition.
- 7. 'at most one case holds': puts the list in a '<>' condition.
- 8. 'at most one case is false': negates them, puts them in a '<>' condition.

As you can guess, version 2 makes use of *negate_cases*, version 4 makes use of *negate_cases_in_boxes* and versions 3 and 4 make use of *disjunct_cases*. I have described most of this earlier, just one thing is new: the "<> −" or "at_most_one-condition" was something I implemented both into *dcg.pl* and into *checker.pl*. The latter was a new sort of task for me, since all my work had concentrated on how to build PRSs from sentences so far. Now the quest was constructing formal statements from given PRSs. Fortunately, I could make use from previous programming done for checking ">< −" or "xor-conditions". Why was this so? The sole logical difference between the statements "precisely one of the cases [...] holds" and "at most one of the cases [...] holds" is that, in the first case, one of the cases listed is true, whereas in the latter cases this isn't necessary. So I basically copied the concept of xor-statement-processing implemented into *checker.pl* and subtracted the statement saying one of the cases are true. What was left was the following: if the condition was

- <>[A,B,C,D,E],

then, recursively, collect the statements

- $A => (neg(B) \wedge (neg(C) \wedge (neg(D) \wedge neg(E))))$,
- $B => (neg(C) \wedge (neg(D) \wedge neg(E)))$,
- $C => (neg(D) \wedge neg(E))$ and
- $D => neg(E)$,

and conjunct them.
I had already modified this part of the predicate so it could cope with an arbitrary number of cases, see Miscellaneous.

## 7 Miscellaneous Work

Here are some minor tasks I did, that do not belong to any of the work areas mentioned:

- I transcribed the examples of Naproche-provable theorems so far into html, so online users can make use of them when working with the web interface. See ex-1.html through ex-3.html.
- Using the simple predicate *rest_list*, I changed the program codes of the existent predicates *lemma* and *theorem* from standard Prolog code to dcg code. This simplifies the code greatly.
- In *checker.pl*, I extended a part of the predicate *check_conditions* that checked exclusive disjunctions, i.e. metasentences of the form "Precisely one of case 1, ... and case n are true". Previously, checker.pl was able to handle up to four cases. I modified the prdicate such that any number n of cases is possible.
- A minor problem in *premises.pl* was fixed by me; I modified the code concerning the implication condition, such that, in the implication PrsA => PrsB, the premises of PrsA plus the conditions of PrsA constitute the premises on which the conditions in PrsB are checked.
- During my internship, Marcos Cramer started the new module *dcg_error.pl*, that checks whether a given input has the right format, disregarding its semantics. To do this, he modified a version of *dcg.pl* that was out-of-date shortly later. I continuously updated *dcg_error.pl*, adding in the new functions that *dcg.pl* had received.
- Up to now, the module *dcg.pl* has grown considerably and become more difficult to read. To make the module's function comprehensible to a larger audience of computer linguists, Marcos Cramer decided it needed to be transcribed into an EBNF (Erweiterte Backus-Naur-Form) format. I studied EBNF notation and made a first version of this transcription (see Appendix).

## 8 Conclusion

I am glad that I decided to do this exact internship. Working in it was quite different from my ordinary studies. It cost effort and a lot of time, it was a very filigree matter to work on and certain tasks were somewhat frustrating. But it was an excellent experience in an area that was very new to me. After a couple of weeks I felt quite satisfied with my progress. I got the rewarding feeling that I was probably doing useful bits of work in the Naproche project. The lectures and workshops titled Formal Mathematics gave me glimpses of the outline and the aims of Naproche. Working fixed hours at a fixed place with the small and select Naproche team turned out to be delightful.

There hardly ever was a dull moment, or at least a lot less than if I had been programming anything on my own. I want to thank Marcos Cramer and Daniel Kühlwein for their help, their patience and, last but not, their good teaching.

# 9 Appendix

## 9.1 Excerpt from *dcg.pl* and *dcg_utils.pl*: the predicate "meta_vp" and its auxiliary predicates

```
%% meta_vp(+PRSList,?Junctor,?MetaPRS,+Mode)
%
% the words in meta_vp are either an affirmative
% or a negative clause.
%
% meta_vp takes the list of case-PRSs from meta_np and does
% one of eight things:
%
% 1. affirmative, and: conjunct them.
% 2. negative, and: negate, then conjuct them.
% 3. affirmative, or: disjunct them.
% 4. negative, or: negate, then disjunct them.
% 5. 'precisely one case holds': puts the list in a '><' condition.
% 6. 'precisely one case is false': negates them, puts them in a '><' condition.
% 7. 'at most one case holds': puts the list in a '<>' condition.
% 8. 'at most one case is false': negates them, puts them in a '<>' condition.
%
% @param: PRSList is a list of case-PRSs
% @param: Junctor can be: and (-> 1), negated_and (-> 2), or (-> 3 or 4),
% xor_rest (-> 5 or 6), at_most_one_rest (-> 7 or 8).
% @param: MetaPRS enters with id and possibly rrefs and exits as a complete PRS.
%
% version 2 makes use of negate_cases/2,
% version 4 makes use of negate_cases_in_boxes/2,
% versions 3&4 make use of dcg_utils:disjunct_cases/2.


% 1. affirmative, and: conjunct PRSes.
meta_vp(PRSList,and,MetaPRS,mode~Mode) -->
correct(mode~Mode),
{
EntryPRS = conds~[]..drefs~[]..mrefs~[],

% merges the list of PRSes
meta_merge(EntryPRS,PRSList,ExitPRS),
!,
ExitPRS = conds~Conds..drefs~Drefs..mrefs~Mrefs..accafter~Acc,
MetaPRS = conds~Conds..drefs~Drefs..mrefs~Mrefs..accafter~Acc
}.

% 2. negative, and: negate, then conjuct them.
```

```
meta_vp(PRSList,negated_and,MetaPRS,mode~Mode) -->
false(mode~Mode),
{

% the list of PRS must be reversed so that the corresponding
% conditions reappear in the correct order.
reverse(PRSList,NewList),

negate_cases(NewList,NegCondList),

% since negate_cases returns a list of conditions, no further
% conjunction is necessary.
MetaPRS = conds~NegCondList..rrefs~[]..drefs~[]..mrefs~[]
}.

% 3. affirmative, or: disjunct them.
meta_vp(PRSList,or,MetaPRS,mode~Mode) -->
correct(mode~Mode),
{
disjunct_cases(PRSList,MetaPRS)
}.

% 4. negative, or: negate, then disjunct them.
meta_vp(PRSList,or,MetaPRS,mode~Mode) -->
false(mode~Mode),
{

% disjunct_cases handles lists of PRSes. hence we need to use
% negate_cases_in_boxes and produce a PRS list.
negate_cases_in_boxes(PRSList,NegPRSList),

disjunct_cases(NegPRSList,MetaPRS)
}.

% 5. 'precisely one case holds': puts the list in a '><' condition.
meta_vp(PRSList,xor,MetaPRS,mode~Mode) -->
correct(mode~Mode),
{

% the ><[..] condition is a grane of syntactic sugar.
% ><[A,B,C] specifically means 'A xor B xor C'.
MetaPRS = conds~[><(PRSList)]..rrefs~[]..drefs~[]..mrefs~[]
}.

% 6. 'precisely one case is false': negates them, puts them in a '><' condition.
```

```
meta_vp(PRSList,xor,MetaPRS,mode~Mode) -->
false(mode~Mode),
{

% ><[..] handles lists of PRSes. hence we need to use
% negate_cases_in_boxes and produce a PRS list.
negate_cases_in_boxes(PRSList,NegPRSList),

% the ><[..] condition is a grane of syntactic sugar.
% ><[A,B,C] specifically means 'A xor B xor C'.
MetaPRS = conds~[><(NegPRSList)]..rrefs~[]..drefs~[]..mrefs~[]
}.

% 7. 'at most one case holds': puts the list in a '<>' condition.
meta_vp(PRSList,at_most_one,MetaPRS,mode~Mode) -->
correct(mode~Mode),
{

% the <>[..] condition is a grane of syntactic sugar.
% <>[A,B,C] specifically means 'at most one of A,B and C holds', or
% 'the cases A, B and C exclude one another'.
MetaPRS = conds~[<>(PRSList)]..rrefs~[]..drefs~[]..mrefs~[]
}.

% 8. 'at most one case is false': negates them, puts them in a '<>' condition.
meta_vp(PRSList,at_most_one,MetaPRS,mode~Mode) -->
false(mode~Mode),
{

% <>[..] handles lists of PRSes. hence we need to use
% negate_cases_in_boxes and produce a PRS list.
negate_cases_in_boxes(PRSList,NegPRSList),

% the <>[..] condition is a grane of syntactic sugar.
% <>[A,B,C] specifically means 'at most one of A,B and C holds', or
% 'the cases A, B and C exclude one another'.
MetaPRS = conds~[<>(NegPRSList)]..rrefs~[]..drefs~[]..mrefs~[]
}.


% correct (+Mode)
%
% contains the affirmative np.

correct(mode~Mode) -->
```

```
verb(_,transitive~copula..mode~Mode),
[correct].

correct(mode~Mode) -->
verb(_,transitive~copula..mode~Mode),
[true].

correct(mode~finite) -->
[hold].

correct(mode~finite) -->
[holds].


% false (+Mode)
%
% contains the negative np.

false(mode~Mode) -->
verb(_,transitive~copula..mode~Mode),
[false].

false(mode~Mode) -->
verb(_,transitive~copula..mode~Mode),
[incorrect].

false(mode~finite) -->
[does,not,hold].

false(mode~finite) -->
[do,not,hold].


% meta_merge (?MergedPRSesSoFar,?PRSesToBeMerged,-ExitPRS)
%
%  uses prs_merge to merge all the PRSes in the List PRSesToBeMerged.

meta_merge(MergedPRSes,[],MergedPRSes).

meta_merge(EntrancePRS,[PRS|Tail],ExitPRS) :-
prs_merge(EntrancePRS,PRS,TempPRS),
meta_merge(TempPRS,Tail,ExitPRS).


% negate_cases(+PRSList,-NegCondList)
```

```
%
%  takes a list of PRSes and makes a list of their negations.
%  this list consists of conditions!

negate_cases([],[]).

negate_cases([PRS|PRSList],[neg(PRS)|NegCondList]) :-
negate_cases(PRSList,NegCondList).


% negate_cases_in_boxes(+PRSList,-NegPRSList)
%
%  takes a list of PRSes and makes a list of PRSes containing their negations.

negate_cases_in_boxes([],[]).

negate_cases_in_boxes([PRS|PRSList],[NegPRS|NegPRSList]) :-
PRS = id~case(Identifier,Id)..accbefore~AccBefore,
NegPRS = id~not_case(Identifier,Id)..conds~[neg(PRS)]..mrefs~[]..drefs~[]
  ..accbefore~AccBefore..accafter~AccBefore..rrefs~[],
negate_cases_in_boxes(PRSList,NegPRSList).

%% add_case_distinction(+CaseList,+In,-Out)
%
% Takes an ingoing PRS and a list of cases, disjuncts the cases and adds the
% resulting PRS to the original PRS's conditions.
%
% Effectively, at least one case on the list must hold.

add_case_distinction(CaseList,In,Out) :-
In = id~Id..accafter~Acc..conds~Conds..drefs~Drefs..mrefs~Mrefs..rrefs~Rrefs,
CaseDistinction = id~case_distinction(Id)..accbefore~Acc,
disjunct_cases(CaseList,CaseDistinction),
Out = id~Id..conds~[CaseDistinction|Conds]..accbefore~Acc..accafter~Acc
  ..drefs~Drefs..mrefs~Mrefs..rrefs~Rrefs.

%% disjunct_cases(+PRSList,?BoxAbove)
%
% takes the PRS list and recursively connects PRSes through "or".
%
% BoxAbove enters with id, accbefore and possibly rrefs and exits as a complete PRS.

disjunct_cases([PRS1,PRS2],BoxAbove) :-
BoxAbove = conds~[PRS1 v PRS2]..drefs~[]..mrefs~[]..accbefore~AccBefore
  ..accafter~AccBefore.
```

```
disjunct_cases([PRS|PRSList],BoxAbove) :-
BoxAbove = id~Id..accbefore~AccBefore,
BoxToTheRight = id~or(Id)..accbefore~AccBefore,
disjunct_cases(PRSList,BoxToTheRight),
!,
BoxAbove = conds~[PRS v BoxToTheRight]..drefs~[]..mrefs~[]..accafter~AccBefore.
```

### 9.2 Excerpt from *dcg.pl*: the predicate ”text“

```
%% text(+In:PRS,-Out:PRS,?GULP)
%
%  Text contains statements, definitions, assumptions, assumption closings, cases
% and case closings. A text can never be empty.
% Adds the content of the parsed text to the In:PRS
%
% @ param GULP type can assume the modi normal, all_ass_closed, theorem and in_case.
% @ param GULP empty can assume the modi no and possible.


text(In,Out,type~Type) -->
[sentence(Id,Content)],
{
statement(In,TmpOut,Id,Content,[]),
!
},
text(TmpOut,Out,type~Type..empty~possible).

text(In,Out,type~Type) -->
definition_text(In,TmpOut),
text(TmpOut,Out,type~Type..empty~possible).

text(In,Out,type~Type) -->
% This rule is for assumptions that don’t get closed by a "thus".
% We could theoretically refrain from checking the absence of a closing "thus",
% as such a thus would make the text_tail fail. But this would produce a huge amount
% of extra progressing power, which would increase exponentially.
% Hence we check the absence of a closing "thus" using the auxiliary predicate
% no_assumption_closing. The reason why this predicate makes the program much more
% efficient, is because it doesn’t do any work (like PRS construction) apart from
% checking the absence of a "thus" that closes this assumption.
{
\+ Type = all_ass_closed
},
```

```
[sentence(Id,Content)],
rest_list(List),
{
no_assumption_closing(List,[]),
In = accafter~InAcc,
AssumptionPRS = id~Id..accbefore~InAcc..rrefs~[],
assumption(AssumptionPRS,Content,[]),
!,
AssumptionPRS = accafter~AssumptionAcc,
TmpConclusionPRS = id~conseq(Id)..accbefore~AssumptionAcc..accafter~AssumptionAcc
  ..mrefs~[]..rrefs~[]..drefs~[]..conds~[]
},
% In this case, the assumption does not get closed, therefore the complete remaining
% PRS construction proceeds in the RHS of the Assumption Condition.
text(TmpConclusionPRS,ConclusionPRS,type~Type..empty~possible),
{
In = id~InId..mrefs~Mrefs..drefs~Drefs..rrefs~Rrefs..conds~Conds..accbefore~Acc
  ..accafter~AccAfter,
Out = id~InId..mrefs~Mrefs..drefs~Drefs..rrefs~Rrefs
  ..conds~[AssumptionPRS ==> ConclusionPRS |Conds]..accbefore~Acc..accafter~AccAfter
}.

text(In,Out,type~Type) -->
% The closed assumption case. For unclosed assumptions, see the previous rule.
assumption_text(In,TmpOut),
text(TmpOut,Out,type~Type..empty~possible).

% during any text, we may introduce cases, close them with a closing statement,
% and continue with text.
text(In,Out,type~Type) -->
cases_text(In,TmpOut,type~Type),
text(TmpOut,Out,type~Type..empty~possible).

% at the end of a proof, however, if we aren't in a case distinction already, we may
% introduce cases and don't need to close them.
text(In,Out,type~Type) -->
{
\+ Type = in_case
},
cases(In,TmpOut,[],CaseList,type~Type..subtype~beginning),
{
% this predicate adds a condition stating that (at least) one of the cases
% that have been introduced must hold.
add_case_distinction(CaseList,TmpOut,Out)
}.
```

```
text(In,Out,type~theorem) -->
lemma(In,TmpOut),
text(TmpOut,Out,type~theorem..empty~no).

text(In,In,empty~possible) --> [].
```

### 9.3 Excerpt from *dcg.pl*: the predicate "cases" with two auxiliary predicates

```
%%  cases(+In,-Out,+CaseListIn,-CaseListOut,?GULP)
%
% a case contains the sentence "case n: Statement" and Text.
% Statement and Text are both transformed into PRSes.
% an implication condition Statement ==> Text is added to In, and
% Statement is added to CaseListIn.
%
% @param In & Out are PRSes,
% @param CaseListIn & CaseListOut are lists of PRSes.
% @param GULP: type enters from text or cases_text and is transferred
% to text or cases itself.
% if type is "in_case", new case distinctions must be announced.
% if type is "rest-list", announcements ("there are N cases") may
% not be made before the case.
%
% Case distinctions within case distinctions are possible, here
% "cases_text" is called by text.

cases(In,Out,CaseListIn,CaseListOut,type~Type..subtype~SubType) -->
case_introduction(type~Type..subtype~SubType),
[sentence(_,[case,Identifier])],
[sentence(CaseRestrId,Content)],
{
In = accafter~Accessibles,
CaseRestr = id~CaseRestrId..accbefore~Accessibles,
proposition_coord(CaseRestr,mode~finite,Content,[]),
!,

% Store Rref Identifier
getval(refids,Refs),
setval(refids,[ref(Content,case(Identifier))|Refs]),

CaseRestr = accafter~Acc,
NewCase = id~text_case(Identifier)..accbefore~Acc..drefs~[]
```

```
  ..mrefs~[]..conds~[]..rrefs~[]..accafter~Acc
},
text(NewCase,CaseScope,type~in_case),
!,
{
In = conds~Conds..id~Id..dref~Drefs..mrefs~Mrefs..accbefore~AccBefore
  ..accafter~AccAfter,
TmpOut = conds~[CaseRestr ==> CaseScope|Conds]..id~Id..dref~Drefs
  ..mrefs~Mrefs..accbefore~AccBefore..accafter~AccAfter,
!
},
cases(TmpOut,Out,[CaseRestr|CaseListIn],CaseListOut,type~Type
  ..subtype~rest-list).

cases(In,In,CaseListIn,CaseListIn,subtype~rest-list) --> [].

% case_introduction contains either a sentence saying "there are N cases" (N>=2)
% or nothing. if type is 'in_case', this sentence is obligatory. on the other hand, if
% we're just presenting the second case, the sentence isn't permitted.

case_introduction(subtype~SubType) -->
{ \+ SubType = rest-list },
[sentence(_,Content)],
{
trigger(type~statement,Content,[there,are,Number,cases]),
number(Number,[Number],[])
}.

case_introduction(type~Type) --> { \+ Type = in_case }, [].

% even within a case, if a subcase A  has been introduced, subcase B
% must not be prefixed by an introductory sentence.

case_introduction(subtype~rest-list) --> [].


% case_closing starts with a case_closing_trigger

case_closing(PRS) --> trigger(type~statement), trigger(type~case_closing),
    proposition_coord(PRS,mode~finite).
```

### 9.4 The Naproche DCG and fo_grammar.pl in EBNF notation

```
(*
```

```
dcg_simple.pl + dcg_lexicon.pl in EBNF-Schreibweise

*)


variable_list = {(math_variable | math_function_term), ","} | math_var_list;

such_that_clause = "such", "that", proposition_coord_finite;

composite_sentence = formula_trigger, math-formula | composite_sentence_finite |
  composite_sentence_infinitive | composite_sentence_to-infinitive;

composite_sentence_finite = sentence_finite | metasentence_finite | sentence_init,
  proposition_coord_that;
composite_sentence_infinitive = sentence_infinitive | metasentence_infinitive;
composite_sentence_to-infinitive = sentence_to-infinitive |
  metasentence_to-infinitive;

sentence_init = "it", "is", ("false" | ["not"], "the", "case");

sentence_finite = np, vp_finite;
sentence_infinitive = np, vp_infinitive;
sentence_to-infinitive = np, vp_to-infinitive;

np = math_formula | specifier, nbar;

specifier = specifier_one | specifier_definite | specifier_indefinite |
  specifier_universal | specifier_negative;

nbar = {adjective}, ([noun], variablebar | noun);

variablebar = math_variable, [such_that_clause];

math_formula = ???????;

(* verb phrases *)

vp_finite = copula_verb_finite , ["not"], (adjective | specifier_indefinite, nbar) |
  "does", "not", vbar_infinitive | vbar_finite;
vp_infinitive = copula_verb_infinitive , ["not"], (adjective |
  specifier_indefinite, nbar) | "does", "not", vbar_infinitive | vbar_finite;
vp_to-infinitive = copula_verb_to-infinitive , ["not"],
  (adjective | specifier_indefinite, nbar) | vbar_finite;

vbar_finite = transitive_verb_finite, np | intransitive_verb_finite;
```

```
vbar_infinitive = transitive_verb_infinitive, np | intransitive_verb_infinitive;
vbar_to-infinitive = transitive_verb_to-infinitive, np |
  intransitive_verb_to-infinitive;


(* words, triggers and lexicon *)


optional_comma = "," | ;

iff = "iff" | "if", "and", "only", "if";

implies = "implies", ["that"];

identifier = {symbol};

noun = "set" | "element" | "number" | "integer" | "class" | "ordinal";

pronoun = "it";

number = "two" | "three" | "four" | "five" | "six" | "seven" | "eight" | "nine" |
  "ten";

adjective = "empty" | "even" | "odd" | "prime" | "positive" | "transitive" |
  "square" | "rational" | "irrational";

transitive_verb_finite = "contains" | "divides";
transitive_verb_infinitive = "contain" | "divide";
transitive_verb_to-infinitive = "to", ("contain" | "divide");

intransitive_verb_finite = "succeeds";
intransitive_verb_infinitive = "succeed";
intransitive_verb_to-infinitive = "to", "succeed";

copula_verb_finite = "is" | "are";
copula_verb_infinitive = "be";
copula_verb_to-infinitive = "to", "be";

specifier_indefinite = "a" | "an" | "some";
specifier_definite = "the";
specifier_one = "precisely", "one";
specifier_negative = "no";
specifier_universal = "every";

universal_quantifier_finite = "for", ("all" | "every");
```

```
existential_quantifier_finite = "there", ("is" | "are" | "exist" ["s"]);
existential_quantifier_infinitive = "there", ("be" | "exist");
existential_quantifier_to-infinitive = "there", "to", ("be" | "exist");

statement_trigger = "then" | "hence" | "therefore" | ["now"],"recall","that" |
  "now",["observe", "that" | "this", ["in", "turn"], "implies"] |
  conseq_trigger | conjunction_trigger;

conseq_trigger = "clearly" | "obviously" | "trivially" | "in", "particular" |
  "observe", "that" | "together", "we", "have" | "furthermore" |
  "this", ["in", turn"], "implies" | "finally" | "also" | ;

conjunction_trigger = "but" | "so" | ;

formula_trigger = "we", ("have" | "get") | ;

ass_trigger_finite = ["now"], ("assume" | "suppose"), ["that"] |
  ["now"], "assume", "for", "a", "contradicition", "that";

ass_trigger_infinitive = ["now"], "let";

ass_trigger_to-infinitive = ["now"], "consider";

variable_declaration_trigger = ["now"], ("consider" | "fix"), ["arbitrary"];

ass_closing_trigger = "thus";

case_closing_trigger = "in", ("all" | "both"), "cases", [","];
```