Apertium-recursive/Formalism

< Apertium-recursive

Contents

File Structure

A `.rtx` file contains attribute categories, tag-order rules, tag-rewrite rules, and reduction rules.

Comments begin with exclamation points (`!`) and end at the end of the line. To include spaces in any name, either escape the space with a backslash (`\`) or enclose the name in double quotes (`"`).

Attribute Categories

An attribute category can be defined like this:

```
gender = m f GD ;
number = sg pl ND ;
```

An attribute category can also specify undefined and default values:

```
gender = (GD m) m f GD;
```

This defines the `gender` category as before, but with the addition that if any rule tries to read the gender of a node that doesn't have a gender tag, the result will be `<GD>` rather than the empty string. It also states that any remaining `<GD>` tags will be replaced with `<m>` tags in the output step.

An attribute category can also specify certain values as non-overwritable.

```
gender = m f @mf;
```

This states that if a lexical unit has a target-language `<mf>` tag and a rule tries to replace that tag with something else, the `<mf>` tag will be used instead of the replacement.

An attribute category can include another:

```
definite = def ind;

! The following are equivalent:
det_type = dem [definite] pos;
det_type = dem def ind pos;
```

The name of an attribute category cannot be any of the following:

Name	Meaning
`lem`	The lemma of an LU or chunk
`lemcase`	The case of the lemma of an LU or chunk
`lemh` and `lemq`	The first and second parts of a multiword with inner inflection. See Multiwords, where lemh corresponds to the inflected portion and lemq is the portion in <g>
`tags`	All of the tags of an LU or chunk
`pos_tag`	The first tag or an LU or chunk
`SIDE_SOURCES`	Conflicts with file directives
`whole`, `chname`, `chcontent`, and `content`	Internal names, some of which are included primarily for compatibility with t*x rules. Do not use these directly

Tag Order

The order of tags for each type of node must be defined like this:

```
n: _.gender.number;
adj: _.gender;
NP: _.number;
```

Where _ represents the part of speech tag. If the part of speech tag is different between the source and target languages, the target language one will be used. The lemma head is automatically appended at the beginning of the pattern and the lemma queue is automatically attached to the end.

To specify a literal tag in a pattern, put it in angle brackets:

```
det: _.<def>.number;
```

Which tag order to use is determined solely by the first tag in the in the pattern of the reduction rule. See the output section below for how to override this choice. See also the macro section for a more powerful version of these rules.

The underscore is not mandatory and is merely a shorthand for `pos_tag`. Thus a literal part of speech tag can be used instead.

In cases where the output of the bilingual dictionary is already correct, the pattern may be replaced by `%`:

```
num: %;
```

If this is the pattern, the output of the node will be the target-language side with no modification whatsoever. Relevant tag-rewrite rules and attribute setting are currently ignored.

Tag Rewrite Rules

This is a way to convert certain sets of tags, either between two languages that have different sets of tenses, or between something like object agreement and number marking.

```
object_agr = o1sg o1pl o2sg o2pl o3sg o3pl ;
number = sg pl ;
person = p1 p2 p3 ;

object_agr > person: o1sg p1, o1pl p1, o2sg p2, o2pl p2, o3sg p3, o3pl p3 ;
object_agr > number: o1sg sg, o1pl pl, o2sg sg, o2pl pl, o3sg sg, o3pl pl ;

VP -> @v NP {2[number=1.object_agr] _1 1} ;
```

In this example, if the verb had `<o2sg>`, it would be converted to `<sg>` when it was set as the `number` attribute of the noun.

```
tense = farpst nearpst pst prs fut nonpst ;

tense > tense: farpst pst, nearpst pst, prs nonpst, fut nonpst ;
```

In this example, no explicit assignment needs to take place and the 4 tenses of the source language (`farpst, nearpst, prs, fut`) would be automatically converted to the 2 of the target language (`pst, nonpst`).

Converting from 4 to 3 with something like

```
tense > tense: farpst pst, nearpst pst ;
```

will also work, the unchanged tags not needing to be explicitly mentioned.

Tags rewrite rules apply in the output step. When building the parse tree, only unconverted tags are used to create chunks. This makes conversions like the following one potentially dangerous:

```
tense > tense: midpst pst, pst pri;
```

If tense is being propagated down through multiple chunks, any `<midpst>` tags will get converted to `<pst>` and then converted again to `<pri>`.

It is also possible to explicitly convert a value, for example when doing comparisons:

```
1.object_agr>number
1.object_agr>person
```

These will clip `object_agr` and convert it to `number` and `person` immediately, regardless of where they are evaluated.

Like attribute category definitions, tag rewrite rules can refer to entire output categories by enclosing them in square brackets. Since the replacement must result in a single value, this can only be done on the source side.

```
pasts = farpst midpst nearpst;
tense = farpst midpst nearpst past pres fut;

! These are equivalent:
tense > tense : [pasts] past;
tense > tense : farpst past, midpst past, nearpst past;
```

Reduction Rules

Reduction rules consist of a node type, an optional name, an optional weight, a pattern, an optional condition, an optional variable setting, and an output, in that order.

```
NP -> det n {2 _ 1};
```

This matches a determiner followed by a noun, combines them into an NP chunk, and at output time produces "noun determiner".

```
NP -> 1: n {1} |
      2: n.*.def {the@det.def.sg _ 1};
```

Here the first rule will match any noun, while the second will match a noun with a `<def>` tag. Since the second rule has a higher weight, the first rule will not be applied if they both match.

```
NP -> NP and@cnjcoo NP [$number=pl] {1 _ 2 _ 3};
```

Here the rule specifies that the resulting chunk will be marked with a `<pl>` tag.

```
AP -> adj and@cnjcoo adj ?(1.gender/sl = 3.gender/sl) {1 _ 2 _ 3};
```

This rule will not apply if the two adjectives have different genders.

The arrow can be written as either `->` or `→`.

A name is written in double quotes before the weight.

```
DP -> "de > gen" NP de@pr NP { 3 + 's@gen _ 1 } ;
```

The process by which rules are selected is described here.

Multiple Outputs

A rule can have multiple outputs, but any non-chunk output cannot be conditioned. A rule with multiple outputs is useful for treating certain tokens as if they occurred in a different order. To write such a rule, list multiple nodes before the arrow and wrap multiple outputs in another set of curly braces (`{}`).

```
DP clitic -> det clitic NP { { 1[number=3.number] _1 3 } _ 2 } ;
DP -> det NP { 1[number=2.number] _ 2 } ;
```

Here the first rule is essentially equivalent to moving the clitic later in the input stream and then applying the second rule.

Multi-output rules are closely related to interpolation, which is described below.

Setting Source or Reference of Chunks

The outputs of a rule by default have empty source and reference sides. However, these can be set in the attribute section using `/sl` and `/ref`.

```
NP -> n [ /sl=1[lem=1.lem/sl]] { 1 } ;
```

If `/sl` or `/ref` is the first thing in attribute section, there must be a space between the bracket and the slash for parsing reasons.

Chunks are always matched by their target sides and source and reference cannot be modified.

Patterns

An element of a pattern must match a single, literal part of speech tag. In order to match multiple part of speech tags, create a separate rule which matches each of them:

```
NOM -> n {1} | np {1};
```

To match a lemma or pseudolemma, place it before the part of speech tag, separated by `@`:

```
NP -> the@det n {2 _ 1};
```

It is also possible to match a category of lemmas:

```
days = sunday monday tuesday wednesday thursday friday saturday;
date -> [days]@n the@det num.ord {2 _ 3 _ 1};
```

Tags besides part of speech can be matched like this:

```
VP -> vbser vblex.pp {1 _ 2};
```

To match a set of tags, enclose the category name in square brackets:

```
non_finite = pp ger;
VP -> vbser vblex.[non_finite] {1 _ 2};
```

Pattern elements can also specify values for the tags of the chunk being output by the rule.

```
number = (ND sg) sg pl sp ND;
NP: _.number;
NP -> n.$number adj {1};
```

This rule specifies that the number tag of the NP chunk should be copied from the noun. It will use the target language side if that is available. If not, it will proceed to the reference side, and then the source side. If all three of these are empty, it will use the default value <ND>. To require that a particular variable be taken from a particular side, put the side after a slash:

```
NP: number;
NP -> det.$number/ref n {1 _ 2};
```

`/sl` refers to the source language, `/tl` to the target language, and `/ref` to anything added by anaphora resolution.

If a pattern element is contributing several tags to the chunk, the following shortcut is available:

```
NP: _.number.gender;
NP -> %n adj {2 _ 1};
```

whatever tags associated with the noun will be copied to the NP chunk as it is

The `%` indicates the noun is the source of all chunk tags not elsewhere specified.

To specify a literal value for a chunk tag, put it in square brackets after the pattern like this:

```
NP: _.gender.number;
NP -> 0: NP cnjcoo NP [$gender=m, $number=pl] {1 _ 2 _ 3} |
      1: NP.f cnjcoo NP.f [$gender=f, $number=pl] {1 _ 2 _ 3} |
      2: NP.*.sg or@cnjcoo NP.*.sg [$gender=m, $number=sg] {1 _ 2 _ 3} |
      3: NP.f.sg or@cnjcoo NP.f.sg [$gender=f, $number=sg] {1 _ 2 _ 3} ;
```

Here we are specifying the conditiones over NP so the match is - two noun phrases and conjucation
NP conjoo NP

That is, treat the gender of the phrase as masculine unless both elements are feminine and the number as singular unless the conjunction is "or" and both elements are singular.

These values can also be conditioned, condensing the above rules to:

```
NP -> NP cnjcoo NP [$gender=(if (1.gender = f and 3.gender = f) f else m),
                    $number=(if (2.lem =cl "or" and 1.number = sg and 3.number = sg) sg else pl)]
                   {1 _ 2 _ 3} ;
```

The pattern only looks at the source language, but it is possible to add constraints:

```
conj_list = and or;
NP: _.gender.number;
NP -> %NP cnjcoo NP ?((2.lem/tl in conj_list) and ~(3.gender = 1.gender)) {1 _ 2 _ 3};
```

This will only match the pattern if it is also the case that the target language lemma of the conjunction is "and" or "or" and the two NPs have different genders. See below for the syntax of conditions.

Piece of pattern	Meaning
`.x`	A literal tag <x>
`.[x]`	Any tag in the category `x`
`.$x`	When building the output chunk for this rule, the value of the `x` attribute should come from this element. Note: this does not match anything. `n.$case.pl` will match <n><pl> but not <n><nom><pl>
`.*`	0 or more arbitrary tags. Note: this contrasts with other places in the pipeline where `*` must match at least 1 tag. A final `.*` is automatically appended to every pattern

So if we even write NP -> n {1}
is it same as NP -> n.* {1}

Outputs

Output elements are written between curly braces and may be any of the following:

Blanks

An underscore represents a single space.

(It used to be that you needed to specify the position of the blank – this is no longer necessary, but the syntax will still be accepted for backwards-compatibility: An underscore followed by a number represents the superblank after that position, so `1 _ 2` is elements 1 and 2 separated by a space while `1 _1 2` is elements 1 and 2 separated by whatever separated them in the input.)

Matched Elements

A number represents the input element in that position with its tags arranged according to the defined output pattern for its part of speech tag. It can be followed by a specification of where those tags should come from.

```
1
! the first input element

1[gender=f]
! the first input element with the gender tag <f>

1[gender=2.gender/ref]
! the first input element with the gender tag of the reference side of the second input element

1[gender=$gender]
! the first input element with the gender tag set to a placeholder to be filled on output with the gender tag of its parent chunk
```

These elements can also be prefixed with `%` to specify that as many tags as possible should be placeholders for tags of the parent chunk.

```
%1[gender=f]
! the first input element with gender tag <f> and all other tags copied from the parent node
```

These elements can be conjoined using +:

```
1[gender=f] + 2
```

This will generate something like `^blah<n><f>+bloop<adj>$`.

Conjoining is currently disallowed if one side is in an if statement and the other is not. It is thus also disallowed if the tag-order rule for either element is a macro.

By default, the order of the output tags is based on the output pattern corresponding to the part of speech tag in the pattern. However, it is possible to override this using parentheses:

```
vblex: _.tense.person.number;
vbinf: _.<inf>;

V -> vblex.inf {1};
  ! result: ^whatever<vblex><inf><{person}><{number}>$

V -> vblex.inf {1(vbinf)};
  ! result: ^whatever<vblex><inf>$
```

The output pattern we need, we can put that here

Titlecase/capitalisation

You can set the lemma case with the lemcase attribute, e.g.

```
N -> %n { 1[lemcase=$lemcase] } ;    ! % will not copy lemcase, have to manually set it
A -> %adj { 1[lemcase=$lemcase] } ;
NP -> A %N { 2[lemcase=1.lemcase] _ 1[lemcase=2.lemcase] } ; ! switch around adj/n lemma case since we switched the words
```

You can also set it to a literal value, e.g. lemcase=aa for lowercase or lemcase=Aa for titlecase

Literal Lexical Units

A new lexical unit can be inserted like this:

```
the@det.def.mf.sp
```

Placeholders can be included using $:

```
the@det.def.$gender.sp
```

Clips from other elements can be placed in square brackets:

```
the@det.def.[2.gender].[3.number/sl]
```

And the case of the lemma can be specified using braces:

```
the@{2.lemcase}.det.def.mf.sp
```

Literal lexical units can also be constructed via the same syntax as matched elements, but with a lemma rather than a number.

```
the(det)[gender=2.gender, number=2.number]
```

When constructed in this way, the tag-order specification is mandatory.

Output Conditionals

An output conditional evaluates a sequence of conditions and outputs the element corresponding to the first one that evaluates to true. The element to be output can be any of the possibilities listed above, the entire chunk, or another conditional.

```
NP -> NP cnjcoo NP
          (if (2.lem/sl = and)
              { 1 _1 3 }
           else
              { 1 _1 2 _2 3 } );
```

Here the rule determines what the final output will be based on the lemma of the conjunction.

```
PP -> DP ?(1.case in might_get_pr)
          (if (1.prep_flag = none)
              { 1 }
           else
              { (if (1.prep_flag = to)
                    to@pr
                 else-if (1.prep_flag = at)
                    at@pr
                 else-if (1.prep_flag = in)
                    in@pr
                 else-if (1.prep_flag = on)
                    on@pr
                 else
                    for@pr
                 )
                 _ 1 } );
```

Here the rule determines first whether to add a preposition. If it is going to add a preposition, it creates a chunk and within that chunk, has another if statement to determine which preposition to add.

The first clause is labeled "if", the last can be "else" or "otherwise", and intermediate ones can be "if", "else-if", or "elif". These labels follow the same rules as logical operators - that is, capitalization, "-", and "_" are all ignored.

For the output of an if statement to have multiple elements, surround those elements with square brackets. Thus the conjunction rule above can be rewritten as follows:

```
NP -> NP cnjcoo NP
         { 1 _1
           (if (2.lem/sl = and)
               [ 2 _2 ]
            else [] )
          3 };
```

Conditions

Conditions are written in parentheses. A condition is a value, an operator, and another value. If the operator is "and" or "or" these values are other conditions, otherwise they are clips or strings. A condition can be negated by writing "not" before the operator.

```
(1.case = 2.case)     ! true if the first and second elements have the same case, otherwise false
(1.case not = 2.case) ! the reverse of the previous line
```

The full list of operations is as follows:

Name	Description	Alternate Spellings
And	Evaluates to true if both arguments evaluates to true, otherwise false	&
Or	Evaluates to true if either argument evaluates to true, otherwise false	|
Equal	Evaluates to true if the arguments are identical strings	=
IsPrefix	Evaluates to true if the right argument occurs at the beginning of the left argument	StartsWith, BeginsWith
IsSuffix	Evaluates to true if the right argument occurs at the end of the left argument	EndsWith
IsSubstring	Evaluates to true if the right argument occurs anywhere in the left argument	Contains
HasPrefix	Evaluates to true if the left argument begins with anything in the list named by the right argument	StartsWithList, BeginsWithList
HasSuffix	Evaluates to true if the left argument ends with anything in the list named by the right argument	EndsWithList
In	Evaluates to true if the left argument is a member of the list named by the right argument	∈

Any of these operators (besides And and Or) can be made to ignore case by adding one of "cl", "caseless", "fold", "foldcase".

```
maybe_get_pr = dat obj;
(1.case in maybe_get_pr)

footwear = boot sock shoe sandal;
```

```
((1.number = du) and (1.lem/tl in_caseless footwear))
! note that "in-case-less", "incl", "IN-cl", and "__IN_CASE_LESS__" would all also work here.
```

Macros

The macro facility is a combination of tag order rules and output conditionals.

```
det_type = dem def ind;

det_dem: _.<dem>.distance;
det_def: _.definite.number;

det: (if (1.det_type = dem)
         1(det_dem)
      else
         1(det_def)
     );
```

Here we define a "det" pattern which will apply the "det_dem" pattern or the "det_def" pattern to its argument based on whether that argument has a `<dem>` tag. Since this is a tag order pattern it will be applied to all `<det>`s by default and can also be manually applied to other things with the `3(det)` syntax.

Macros are only allowed to clip from the input node (referred to as `1`), including any values passed in.

If a macro specifies a value for an attribute, it will override anything that is passed in. Thus if the above example had `1(det_dem)[distance=prx]` rather than `1(det_dem)`, invoking it as `2(det)[distance=dist]` and as `2(det)[distance=med]` would make no difference and the output would be have `<prx>` regardless.

A macro is not required to function as a tag order pattern and may output anything or nothing, so long as it only accesses attributes of the input node.

If you want to call a macro and what node gets passed in doesn't matter, you can use the symbol `*` to represent an empty node.

```
maybe_det: (if (1.definite = def)
               [the@det.def.sp _]
            elif (1.number = sg)
               [a@det.ind.sp _]
            else [] );
DP -> n { *(maybe_det)[number=1.number, definite=$definite] 1 };
```

This will insert "the" if the DP is definite, "a" if it's indefinite and singular, and will output only the noun otherwise.

Since a macro needs to be contained in a conditional but not all macros are conditional, they permit the keyword `always` in addition to `if`, `else`, etc.

```
vaux: (always 1(vblex));
```

This macro is essentially an alias of `vblex`.

Interpolation

Sometimes it is necessary to insert words into existing nodes, such as when generating certain clitics.

```
NP -> adj n { 2 _1 1 } ;
DP -> det NP (if ($lu-count = "2")
                 { 1 _1 2 }
              else
                 { 1 _ >3 _1 2 } ) ;
VP -> DP v.pprs { 1 < be(vaux) _1 2 } ;
```

These rules represent a scenario where target language present progressive is marked with a clitic which is placed between a determiner and noun phrase. At the `VP` level, the clitic is created and inserted into the `DP` with a less-than sign (`<`). Then at the `DP` level, the rule checks whether anything has been inserted by checking whether the value of `$lu-count` is `2`, which it would be if nothing had been inserted. If `$lu-count` is not `2`, then the inserted item is output in the appropriate place.

The inserted value is referred to as `>3` rather than `3` to tell the compiler that it is an inserted value so as to prevent error messages about trying to access a node that doesn't exist.

Some possible input and output from these rules (written monolingually for simplicity):

```
^the<det>$ ^green<adj>$ ^frog<n>$ ^speak<v><pprs>$

the NP[green frog] speak
DP[the NP[green frog]] speak
VP[DP[the NP[green frog]] speak]

DP[the NP[green frog] be] speak
the be NP[green frog] speak
the be frog green speak

^the<det>$ ^be<vaux>$ ^frog<n>$ ^green<adj>$ ^speak<v>$
```

This is functions as the inverse of rules with multiple outputs. The reverse of the above rules could be something like this:

```
NP -> n adj { 2 _1 1 } ;
DP -> det NP { 1 _1 2 } ;
```

```
DP vaux -> det vaux NP { { 1 _1 3 } _2 2 } ;
VP -> DP vaux v { 1 _1 3[tense=pprs] } ;
```

Multi-output rules take things inside and moves them out, while interpolation takes things outside and moves them in.

Global Chunk Variables

For passing nodes up and down a tree, an alternative to multi-output rules and interpolation is global chunk variables. Global chunk variables are referred to with double dollar signs and are set in the attribute literal section of a rule.

```
VP -> %vblex DP.$itg [$$wh_word=(if (2.itg = itg) 2)] { 1 (if (2.itg not = itg) [ _ 2 ]) } ;
```

The value of this variable can then be included in the output step of any rule.

```
S -> DP.nom VP { (if (2.itg = itg) [ $$wh_word _ ] ) 1 _ 2 } ;
```

If a rule attempts to output an unset variable, the result will be no output. All variables are reset at the end of the output step.

Global String Variables

Global string variables function much like global chunk variables, but as clips, rather than LUs. They are referred to with $%.

```
Subj -> NP.nom [$%subj_number=1.number] { %1 } ;
```

The value can then be used in any subsequent rule prior to the end of the output step.

```
V -> vblex { 1[number=$%subj_number] } ;
```

Clips

When clipping a tag or lemma from a lexical unit in the input stream, `/sl` refers to source language, `/tl` refers to target language, and `/ref` refers to the output of apertium-anaphora. Chunks, meanwhile, have only 1 side, which is `/tl`. If the side is left unspecified, then `/tl` will be clipped. However, if an LU is being clipped from and the value for `/tl` is empty or is the unspecified value for that attribute category, it will try again with `/ref` and then with `/sl`. The order in which sides are checked in the case of unspecified values can be modified by a `SIDE_SOURCES` directive (see below).

Almost anywhere that a clip or a literal value is used as a value, it can be replaced with an if statement using the same syntax as output conditionals and macros.

```
VP -> v vaux [$negative=(if (1.negative = neg or 2.negative = neg) neg else pos)]
             { 2 _1 1[tense=(if (2.lem in verb_ing) pprs) else inf)] } ;
```

The one exception is embedding an if statement inside a conditional: `(x in (if ...))`.

File Directives

File directives control the behavior of compilation. They are set using the same syntax as attribute categories. For example:

```
SIDE_SOURCES = tl ref ;
```

Directive	Values	Default	Description
`SIDE_SOURCES`	1 or more of `sl`, `tl`, and `ref`	`tl ref sl`	specifies the order in which unlabeled clips check the sides of an LU

Brackets

A summary of which means what where:

Bracket	General Meaning	Uses	Examples	Comments
()	Condition	If statement	`(if ...)`	
		Condition	`(a in b)`	in an if statement
		Pattern condition	`NP NP ?(1.case = 2.case)`	
		Pattern override	`1(vb_impers)`	use the vb_impers output rule rather than the output rule chosen based on the pattern
		Macro invocation	`*(maybe_det)`	
		Attribute defaults	`gender = (GD m) m f mf GD;`	
[]	List	Rule variable setting	`[$tense=past, $number=sg]`	
		Node variable setting	`1[tense=2.tense, number=2.number]`	
		Set inclusion	`tense = [finite] inf ger;`	tense is composed of <inf>, <ger> and everything in finite
		Group tag rewriting	`poss > number : [poss_sg] sg, [poss_pl] pl;`	
		Grouping in if statements	`(if (whatever) [1 _ 2])`	
		Pattern tag sets	`NP.*.[case_not_nomacc]`	
		Pattern lemma sets	`[days]@n`	
{}	Chunk	Output	`NP -> n { 1 } ;`	
		Chunk	`NP clitic -> det clitic NP { { 1 _ 3 } _ 2 } ;`	

Correspondence with t*x

```
number = sg pl;
gender = m f;
pre_adj = gran buen;

n: _.gender.number;
adj: _.gender;
NP: _.number;

NP -> adj n.$number ?(1.number = 2.number)
          (if (1.lem/tl incl pre_adj)
              {1[gender=2.gender] _1 2}
           else
              {2 _1 1[gender=2.gender]}
          ) ;
```

```
<transfer>
 <section-def-cats>
  <def-cat "n">
   <cat-item tags="n"/>
   <cat-item tags="n.*"/>
  </def-cat>
  <def-cat "adj">
   <cat-item tags="adj"/>
   <cat-item tags="adj.*"/>
  </def-cat>
 </section-def-cats>
 <section-def-attrs>
  <def-attr n="number">
   <attr-item tags="sg"/>
   <attr-item tags="pl"/>
  </def-attr>
  <def-attr n="gender">
   <attr-item tags="m"/>
   <attr-item tags="f"/>
  </def-attr>
 </section-def-attrs>
 <section-def-lists>
  <def-list n="pre_adj">
   <list-item v="gran"/>
   <list-item v="buen"/>
  </def-list>
 </section-def-lists>
 <section-rules>
  <rule comment="adj n">
   <pattern>
    <pattern-item n="adj"/>
    <pattern-item n="n"/>
   </pattern>
   <action>
    <choose>
     <when>
      <test>
       <not>
        <equal>
         <clip pos="1" side="tl" part="number"/>
         <clip pos="2" side="tl" part="number"/>
        </equal>
       </not>
      </test>
      <reject-current-rule/>
     </when>
    </choose>
    <choose>
     <when>
      <test>
       <in caseless="yes">
        <clip pos="1" side="tl" part="lem"/>
        <list n="pre_adj"/>
       </in>
```

```
      </test>
      <out>
       <chunk name="default">
        <tags>
         <tag><lit-tag v="NP"/></tag>
        </tags>
        <lu>
         <clip pos="1" side="tl" part="lemh"/>
         <lit-tag v="adj"/>
         <clip pos="2" side="tl" part="gender"/>
        </lu>

        <lu>
         <clip pos="2" side="tl" part="lemh"/>
         <lit-tag v="n"/>
         <clip pos="2" side="tl" part="gender"/>
         <clip pos="2" side="tl" part="number"/>
        </lu>
       </chunk>
      </out>
     </when>
     <otherwise>
      <out>
       <chunk name="default">
        <tags>
         <tag><lit-tag v="NP"/></tag>
        </tags>
        <lu>
         <clip pos="1" side="tl" part="lemh"/>
         <lit-tag v="adj"/>
         <clip pos="2" side="tl" part="gender"/>
         <clip pos="1" side="tl" part="lemq"/>
        </lu>

        <lu>
         <clip pos="2" side="tl" part="lemh"/>
         <lit-tag v="n"/>
         <clip pos="2" side="tl" part="gender"/>
         <clip pos="2" side="tl" part="number"/>
         <clip pos="2" side="tl" part="lemq"/>
        </lu>
       </chunk>
      </out>
     </otherwise>
    </choose>
   </action>
  </rule>
 </section-rules>
</transfer>
```

`number = sg pl;`	`<def-attr n="number"> <attr-item tags="sg"/> <attr-item tags="pl"/> </def-attr> <def-list n="number"> <list-item v="sg"/> <list-item v="pl"/> </def-list>`	It isn't shown in the above example, but each list simultaneously defines an attribute category and a list.
`n: _.gender.number;`	(no direct equivalent)	
`NP ->`	`<tags> <tag><lit-tag v="NP"/> </tag> ... </tags>`	The further contents of `<tags>` is determined by `NP: _.number;`, which indicates that those contents will be a `number` tag, probably clipped from one of the inputs.
`n`	`<def-cat n="some_unique_name"> <cat-item tags="n"/> <cat-item tags="n.*"/> </def-cat> ... <pattern-item n="some_unique_name"/>`	
`.$number`	`<clip pos="2" side="tl" part="number"/>`	This determines the contents of `<tags>` in the output chunk.
`?`	`<choose> <when> <test> <not> ... </not> </test>`	There is no functionality equivalent to `<reject-current-rule shifting="yes"/>`.

	<reject-current-rule/> </when> </choose>	
1.number	<clip pos="1" part="number"/>	In the example rule, the clips are written as being side="tl", but an unspecified clip will actually check all three sides (target, then reference, then source) until it finds a value.
(if (...) ... else ...)	<choose> <when> <test> ... </test> <out> ... </out> </when> <otherwise> <out> ... </out> </otherwise> </choose>	
(... incl ...)	<in caseless="yes"> ... <list n="..."/> </in>	
_1	<b pos="1"/>	
1[gender=2.gender]	<lu> <clip pos="1" side="tl" part="lemh"/> <lit-tag v="adj"/> <clip pos="2" side="tl" part="gender"/> <clip pos="1" side="tl" part="lemq"/> </lu>	<lit-tag v="adj"/> should actually be <clip pos="1" side="tl" part="pos_tag"/> where pos_tag is a special attribute that returns whatever the first tag is.

`{ ... }`	`<chunk name="default">`
	`...`
	`</chunk>`
		It is possible to make the name be something other than default, for example with `n.$lem/sl` in the pattern.

Technically this would compile to a rule which outputs an `NP` chunk containing the input unchanged and also a separate postchunk rule that would do the actual rearranging so that the conditionals can depend on changed values of the chunk tags.
