A long introduction to transfer rules

En français

Writing transfer rules seems to be tricky. People generally understand the basic concepts, but they struggle with the formalism. We think the formalism isn't that bad. And compared to many other formalisms,[1] it's fairly straightforward. Maybe one of the reasons people struggle is that we mix declarative and procedural programming. Could be.

Contents

Some formalities

Before starting, it is important to give some idea of what we can't do, before explaining what we can. If you come at rule-learning expecting something else, then it's likely to be confusing.

- There are no recursive rules. Rules match fixed-length patterns. There is no optionality at the level of words. There is no way of saying one-or-more, it's just one.
  - (but see Apertium-recursive if you're starting a new language pair)
- Apertium's rules are very tied to the Apertium stream format. If you don't understand the stream format, it will be a lot more difficult to understand the rules.
- Rules contain both *declarative* parts and *procedural* parts. You can't just expect to say *what* you want or *how* you want to do it. You need to do both -- but in different places (but it's quite intuitive).
- Patterns match only on the source side. Not on the target side.
- The structural transfer has no access to the information in the target language morphological generator. This means that if the transfer needs some information about the available forms of a particular word, e.g. if it is only singular, or only plural, then this information needs to go in the bilingual dictionary.

Approaching the process of writing transfer rules

- Think bottom up, not top down... start with questions like "what is the simplest and least bad equivalent for dative in a language which does not have a dative?" not "how can I change SOV order to SVO order".

Lexical transfer and structural transfer

See also: Bilingual dictionaries

See also

At this point it's worth not confusing the roles of lexical transfer and structural transfer. There is a grey area between the two, but there are also big parts that don't overlap.

- **Lexical transfer** (the `lang1-lang2.dix` file):
  - Nearly always gives translations between words, not tags.
  - Can add or change tags, on a per-lemma basis.
  - Doesn't do reordering.
  - Can be used to give a *head's up* to the structural transfer to draw attention to missing features, or features that cannot be decided on a no-context basis. For example:
    - the `<ND>` and `<GD>` tags which say "Hey, when I'm translating this word I don't know what the gender or number should be -- structural transfer! I need your help to find out", or
    - the `<sint>`[2] tag which says "¡Ojo! if you're writing a transfer rule for adjectives, and it matches this adjective then you need to think about how you're going to handle the comparative and superlative forms"
- **Structural transfer** (the `lang1-lang2.t*x` files):
  - Rarely gives translations between single words.
  - Often adds or changes tags on a per-category (groups of lemmas) basis.
  - Can change the order of words.

A rule-of-thumb is that if the rule applies to all words in a category, it probably wants to be treated in the structural transfer, and if it applies to just part of those words, then maybe it needs to be dealt with in the lexical transfer.

The output of the lexical transfer looks like this: from dix, direct translation

```
^slword<sometag>/tlword<sometag>$  ^slword1<sometag><blah>/tlword3<sometag><foo>$ ^slword3<sometag><blah>/tlword2<sometag><GD>$
```

Where the output of the structural transfer would look like this:

```
^tlword<sometag>$ ^tlword3<sometag><foo>$ ^tlword2<sometag><GD>$
```

That is, when you are in the first structural transfer stage you have access to both the source and target sides of the translation. After the first structural transfer stage, you only have access to the target side.

How lexical transfer is processed

In my case i am keeping primary pos tags in my bilingual dict so i will get those tags along with the tags of source lemma

Given a input lexical unit: `^slword<tag1><tag2><tag3>$`

If we have the following in the bilingual dictionary:

```
    <e><p><l>slword<s n="tag1"/><s n="tag2"/></l><r>tlword<s n="tag1"/></r></p></e>
```

We will get this output from the lexical-transfer module:

```
  ^slword<tag1><tag2><tag3>/tlword<tag1><tag3>$
```

Note that the target-language lexical form as defined in the bilingual dictionary is produced by replacing two tags on the source side (i.e. `<s n="tag1"/><s n="tag2"/>`) with one tag on the target side (i.e. `<s n="tag1"/>`). Important: any source-language tags not matched in the bilingual dictionary entry are copied into the output on the target language side. In our example `<tag3>` gets copied from the input to the output.

Some preliminaries

- Pattern: What we match. Patterns are given numbers based on what their order was, so the first pattern is pos="1", second is pos="2", etc.
- Action: What we do with the matched pattern, including outputting it.

Overview of a transfer file

It's hard to give a step-by-step overview of what a transfer file looks like because there is quite a lot of obligatory parts that need to go into even the most basic file. But, it's important to get a general view before we go into the details. Here is an example in which I'm deliberately not going to use linguistic names for the different parts, to try and avoid assumptions.

```
<?xml version="1.0" encoding="utf-8"?>
<transfer>
  <section-def-cats>
    <def-cat n="some_word_category">
      <cat-item tags="mytag.*"/>
    </def-cat>
  </section-def-cats>
  <section-def-attrs>
    <def-attr n="some_feature_of_a_word">
      <attr-item tags="myfeature"/>
      <attr-item tags="myotherfeature"/>
    </def-attr>
  </section-def-attrs>
  <section-def-vars>
    <def-var n="blank"/>
  </section-def-vars>
  <section-rules>
    <rule>
      <pattern>
        <pattern-item n="some_word_category"/>
      </pattern>
      <action>
        <let><clip pos="1" side="tl" part="some_feature_of_a_word"/><lit-tag v="myotherfeature"/></let>
        <out>
          <lu><clip pos="1" side="tl" part="whole"/></lu>
        </out>
      </action>
    </rule>
  </section-rules>
</transfer>
```

to change the target side tag, the match part tag will be changed to “myotherfeature” always

I'll try and give a tag-by-tag account. (Confusingly, tags can be <XML> tags like the above or they can be linguistic tags in the input like <adv>, both types surrounded by < and >; I try to explicitly call them "XML tags" or "input tags" here.)

The <transfer> and </transfer> XML tags don't do anything. They just encapsulate the rest of the sections.

The <section-def-cats> contains definitions of pattern categories that can be **matched** by rules. Each <def-cat> has a name in the attribute n and a list of <cat-item>'s with the input patterns it should match. A pattern can be defined by tags or lemma or both. The tags attribute often contains a .* meaning "one or more arbitrary input tags".

The <section-def-attrs> contains definitions of input tag sets that can be **used** (read to, changed or output) by rules (using the <clip> feature). Each <def-attr> has a name in the attribute n and a list of <attr-item>'s each with a list of consecutive tags.

The <section-def-vars> contains variable names that can be used in rules. Each <def-var> has a name in the attribute n.

The follows the rules in <section-rules>, each rule is inside <rule>..</rule> and has first a <pattern> with a list of one or more <pattern-item>'s each referring to the name of a <cat-item>. This is used for *matching* on the input. Then follows the <action> which specifies what to do with the matched words.

In this example, we use <let> to change the target language side (side="tl") part of the first matched pattern (<clip> with pos=1), changing the input tags that match the <attr-item> with name "some_feature_of_a_word" to contain the input tag <myotherfeature> (using <lit-tag> with that value, given by the attribute v). So the attribute "some_feature_of_a_word" can match either the input tag <myfeature> or <myotherfeature>, but in the output it will only be <myotherfeature>. (If the input tag wasn't there, nothing is changed). Finally, after changing that input tag, the rule uses <out> to print the matched and changed pattern. The <lu> ensures the pattern is surrounded by ^ and $. The <clip pos="1" side="tl" part="whole"> refers to the whole lemma + tag sequence of the target language side of the first matched pattern.

How rules are applied

Rules are tried from top to bottom in the file, so first rules are prioritised. But the longest match (reading from left to right) is used (LRLM), so longer matches, ie. matching more `<pattern-item>`s, will be prioritised over shorter.

Practical examples

1-level structural transfer: basics

Input:	*Otišla si*	*tiho*	*i*	*bez*	*pozdrava*
Output:	*You left*	*quietly*	*and*	*without*	*a word*

Lexical transfer

For the purposes of this example, we are going to use some "pre-prepared" lexical transfer output. In a real system, this would be the output from running `lt-proc -b` on the bilingual dictionary of the sytem. So, we pass the phrase through our "imaginary" lexical-transfer stage, and come up with the following output:

```
^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$
^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$
^tiho<adv>/quietly<adv>$
^i<cnjcoo>/and<cnjcoo>$
^bez<pr>/without<pr>$
^pozdrav<n><mi><sg><gen>/word<n><sg><gen>$
```

Put the output into a file called `example1.txt` we're going to be needing it later.

At this point if you haven't already, it's worth trying *model zero*, that is no transfer rules, so try passing the output through the transfer file we made earlier (see here -- save it in a file called `rules.t1x`) and generating it:

First, we need to compile:

```
$ apertium-preprocess-transfer rules.t1x rules.bin
```

Then we can pass through transfer:

```
$ cat /tmp/example1.txt | apertium-transfer -b rules.t1x rules.bin
^leave<vblex><lp><f><sg>$
^be<vbser><clt><pres><p2><sg>$
^quietly<adv>$
^and<cnjcoo>$
^without<pr>$
^word<n><sg><gen>$
```

Then try and pass it through a generator of English (which can be compiled from any `en.dix` file):[3]

```
$ cat /tmp/example1.txt | apertium-transfer -b rules.t1x rules.bin | lt-proc -g sh-en.autogen.bin
#leave
#be
quietly
and
without
#word
```

This is obviously inadequate, but don't worry, we're going to use the structural transfer module to make it adequate!

Thinking it through

Let's think about what changes we need to make in order to convert this into an adequate form for target language generation. NB: If we want to change information, it's a procedure, if we want to output it or not, it's a declaration.

Procedures

1. If the source language tag is `<lp>`, change the target language tag to `<past>`

Declarations

1. Output a subject pronoun which takes its person and number information from the main verb.
2. Output the main verb with information on category and tense (but not gender and number).
3. Do not output the auxiliary verb (*biti*, "be").
4. Output nouns with category and number (but not case).
5. Words should be output encapsulated in ^ and $
6. Tags should be output encapsulated in < and >

Work order

So, what order do we do these in, well it doesn't really matter -- an experienced developer would probably do it in two stages, but for pedagogical purposes, we're going to split it up into five stages:

- First we're going to write a rule which matches the lp and auxiliary construction, and outputs only the main verb (declarations: 2, 3, 5)
  - Define the categories of "lp" and "auxiliary"
- Second we're going to edit that rule to change the source language tag from `<lp>` to `<past>` (procedure: 1)
  - Define the attribute of "tense"
- Third we're going to edit the same rule to not output gender and number (declaration: 2)
  - Define the attribute of "verb_type"
- Fourth we're going to edit the same rule to output a subject pronoun before the verb (declarations: 1, 5, 6)
  - Define the attributes of "person" and "number"
- Fifth we're going to write a new rule which matches the noun construction and output only category and number (declarations: 4, 5)
  - Define the category of "noun"
  - Define the attribute of "noun_type"

Cheatsheet

Here is what the input and output of each of the changes above will look like.

Change	Input	Output

1	`^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$`
	`^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$`
		`^leave<vblex><lp><f><sg>$`

2	`^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$`
	`^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$`
		`^leave<vblex><past><f><sg>$`

3	`^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$`
	`^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$`
		`^leave<vblex><past>$`

4	`^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$`
	`^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$`
		`^prpers<prn><subj><p2><mf><sg>$`
		`^leave<vblex><past>$`

5	`^pozdrav<n><mi><sg><gen>/word<n><sg><gen>$`	`^word<n><sg>$`

Implementation

Step 1

We're working on the output (i.e., on the first two lines of *example1.txt*):

```
^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$
^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$
```

1. Make our categories in *rules.t1x*. Replace the dummy def-cat with:

```
  <section-def-cats>
    <def-cat n="lp">
      <cat-item tags="vblex.*.*.lp.*"/>
    </def-cat>
    <def-cat n="biti-clt">
      <cat-item tags="vbser.clt.pres.*"/>
    </def-cat>
  </section-def-cats>
```

Why do we need `.*.*` ? -- Because of how the matching system in categories works. In the middle of tag sequences, a `*` is counted as a single tag. At the end, it is counted as any sequence of tags. So, we have `<vblex>` followed by any tag, followed by any tag, followed by `<lp>` followed by any sequence of tags. People familiar with regular expressions should take a special note that `.*` does not mean "anything", and "*" does not mean "any amount of times" (except at the end of a pattern). `*` is more like a wildcard.

2. Edit the example rule (notice that the example rule outputs only the first among two tokens, so that our biti-clt token will disappear) and replace the pattern.

```
      <pattern>
        <pattern-item n="lp"/>
        <pattern-item n="biti-clt"/>
      </pattern>
```

3. Save and compile the rule file.

```
$ apertium-preprocess-transfer rules.t1x rules.bin
```

Now test it:

```
$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin

^leave<vblex><lp><f><sg>$
^quietly<adv>$
^and<cnjcoo>$
^without<pr>$
```

```
^word<n><sg><gen>$
```

Great!

Step 2

1. Add a `<def-attr>` giving the possible values of the `tense` feature.

So, now we've got rid of the verb "to be", we want to change the tag for tense from `<lp>` to `<past>`. This will involve using a statement, explicitly telling the transfer what we want to change. But before that, we need to define which possible values our feature can take. We do this with a `<def-attr>`. Let's call it `tense`. Add the following to the `<section-def-attrs>`.

```
    <def-attr n="tense">
      <attr-item tags="lp"/>
      <attr-item tags="past"/>
    </def-attr>
```

This says that words can have an attribute `tense`, which may be one of two values, either `<lp>` or `<past>`. As an exercise to the reader, you could try and add some other possible values.

2. Write a statement which changes the tense on the target-language (`side="tl"`)

I can use let to change any tag from sumerian to english

This statement should go immediately after the `<action>` tag and before the `<out>` tag.

```
      <action>
        <let><clip pos="1" side="tl" part="tense"/><lit-tag v="past"/></let>
        <out>
```

(Note: This will *always* change the target language tag to `<past>`, even if the input is not `<lp>`. This is good enough for our current example, but for a description of how conditional statements work, stay tuned!)

Now test it:

```
$ apertium-preprocess-transfer rules.t1x rules.bin

$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin
^leave<vblex><past><f><sg>$
^quietly<adv>$
^and<cnjcoo>$
^without<pr>$
```

```
^word<n><sg><gen>$
```

Step 3

So far we have been outputting the whole target-language lexical unit. The reader will recall that the target-language lexical form is produced by passing the output of the source language part-of-speech tagger through the lexical-transfer module, and replacing a prefix on the source code with a prefix on the target side. Any source-language tags not matched by the prefix in the bilingual dictionary are copied into the output on the target language side.

This is why we have:

```
^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$
```

Our input from the source language is `otići<vblex><perf><iv><lp><f><sg>`, our bilingual dictionary maps `otići<vblex><perf><iv> → leave<vblex>` so we are left with the tags `<f><sg>` on the target-language side. These tags are not needed for generation of English, so we need to declare that they should not be output.

1. Define the features that we want to output.

We have already defined the feature (`def-attr`) `tense`, the only remaining tag we need to output is the verb type. So, make a new `def-attr` for verb type. In Apertium usually, these are prefixed with a_ this stands for "attribute", so it isn't confused with other uses of "verb".

```
    <def-attr n="a_verb">
      <attr-item tags="vblex"/>
      <attr-item tags="vbser"/>
    </def-attr>
```

Note: The reason we have two tags here, is that in many language pairs, the verb "to be" (in this case, *biti*, but also *ser*, *être* etc.) is tagged differently from other verbs due to its morphological and syntactic behaviour.

In sumerian also different meaning of verb

2. Declare which features we want to output.

So, now that we've got our two features defined, let's change our `<out>` statement:

Where we previously had:

```
          <lu><clip pos="1" side="tl" part="whole"/></lu>
```

Replace it with:

```
          <lu>
            <clip pos="1" side="tl" part="lem"/>
            <clip pos="1" side="tl" part="a_verb"/>
            <clip pos="1" side="tl" part="tense"/>
          </lu>
```

The two second features we know, as we've just added them. The "lem" feature is a pre-defined feature which corresponds to "lemma". There are others, if you remember "whole" corresponds to the whole lexical unit, and "tags" corresponds to all the tags.

Now test it:

```
$ apertium-preprocess-transferb rules.t1x rules.bin

$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin
^leave<vblex><past>$
^quietly<adv>$
^and<cnjcoo>$
^without<pr>$
^word<n><sg><gen>$
```

Woo! What just happened was that we magically deleted the two unwanted tags `<f>` and `<sg>` by *not specifying that we wanted them*. Tag deletion is thus not stated as tag deletion `tag → 0`, but by not declaring them.

Now we should have a lexical form for "leave" which we can generate... let's give it a go:

```
$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin | lt-proc -g sh-en.autogen.bin
left
quietly
and
without
#word
```

We're getting there, now let's move onto the next step...

Step 4

To recap, our input is:

```
^otići<vblex><perf><iv><lp><f><sg>/leave<vblex><lp><f><sg>$
^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$
```

And our current output is:

```
^leave<vblex><past>$
```

What we need to do is take the information from the (now not output) verb *biti* "to be" and use it to output a personal pronoun before the verb. Remember, that when we do not output something we are not *deleting* it, it is still there in the input, we are just choosing not to put it in the output.

The string that we want to output looks like this:

```
^prpers<prn><subj><p2><mf><sg>$
```

1. Output a skeleton pronoun.

So, we need to take the person and number information from the verb "to be", and the rest we need to just declare to be output. Let's start with outputting the string `^prpers<prn><subj>$` before the verb.

```
            <lu>
              <lit v="prpers"/>
              <lit-tag v="prn.subj"/>
            </lu>
            <b/>
```

The `<lu>` outputs a `^` and the `</lu>` outputs a `$`. The `<lit>` tag outputs a string literal, and the `<lit-tag>` outputs a tag sequence which starts with a <, ends with a > and where each `.` is replaced with ><. The `<b/>` outputs a single space `' '` character.

2. Define features for person and number.

The next step is to declare the features from the verb be output. To do this, we need to again define some features and their possible values.

```
      <def-attr n="person">
        <attr-item tags="p1"/>
        <attr-item tags="p2"/>
        <attr-item tags="p3"/>
      </def-attr>
```

```
    <def-attr n="number">
      <attr-item tags="sg"/>
      <attr-item tags="pl"/>
    </def-attr>
```

We then go back to our skeleton pronoun and add the missing features:

```
          <lu>
            <lit v="prpers"/>
            <lit-tag v="prn.subj"/>
            <clip pos="2" side="sl" part="person"/>
            <lit-tag v="mf"/>
            <clip pos="2" side="sl" part="number"/>
          </lu>
          <b/>
```

There are two main things worth noting here, the first is that we are "clipping" (copying a part of a string which matches a pattern) from position 2, that is the verb "to be".

To be clearer, the statement `<clip pos="2" side="sl" part="person"/>` makes a copy of the part of the lexical unit in position 2, on the source language side, which matches one of the patterns defined in the `<def-attr>` for "person" (which can be either `<p1>`, `<p2>` or `<p3>`). If we look at the lexical transfer output for position "2" this will become clearer:

```
^biti<vbser><clt><pres><p2><sg>/be<vbser><clt><pres><p2><sg>$
 |                      |__|    | |                           |
 |                      person  | |                           |
 |______________________________| |___________________________|
          side="sl"                       side="tl"
```

If we tried to clip from position 1, we would get no result for the "person" attribute because the verb in position 1 does not contain any tags which match the tags in the `def-attr` for `person`. The second is that we've outputted the gender as a literal tag. This is because although our first verb gives us the gender, it gives us just `<f>`, and English doesn't express gender in the second person pronoun.

Now test it:

```
$ apertium-preprocess-transfer rules.t1x rules.bin

$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin
^prpers<prn><subj><p2><mf><sg>$ ^leave<vblex><past>$
^quietly<adv>$
^and<cnjcoo>$
^without<pr>$
^word<n><sg><gen>$
```

And then with the generator:

```
$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin | lt-proc -g  sh-en.autogen.bin
you left
quietly
and
without
#word
```

Step 5

So, the last remaining thing to do is to not output the genitive tag on the noun. We're going to have to make a whole new rule to match nouns.

1. Define a category for nouns

```
    <def-cat n="noun">
      <cat-item tags="n.*"/>
    </def-cat>
```

2. Make a new rule which matches nouns as defined by the previous category.

To start with we'll just output the `whole` lexical unit.

```
<rule>
  <pattern>
    <pattern-item n="noun"/>
  </pattern>
  <action>
    <out>
      <lu><clip pos="1" side="tl" part="whole"/></lu>
    </out>
  </action>
</rule>
```

3. Define a new category for noun type

```
    <def-attr n="a_noun">
      <attr-item tags="n"/>
      <attr-item tags="np"/>
    </def-attr>
```

4. Adjust the rule to output only noun type and number

Remember that we previously defined the `<def-attr>` for "number".

```
    <out>
      <lu>
        <clip pos="1" side="tl" part="lem"/>
        <clip pos="1" side="tl" part="a_noun"/>
        <clip pos="1" side="tl" part="number"/>
      </lu>
    </out>
```

Now recompile, and test it:

```
$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin
^prpers<prn><subj><p2><mf><sg>$ ^leave<vblex><past>$
^quietly<adv>$
^and<cnjcoo>$
^without<pr>$
^word<n><sg>$

$ cat example1.txt | apertium-transfer -b rules.t1x rules.bin  | lt-proc -g sh-en.autogen.bin
you left
quietly
and
without
word
```

Success!

Step 6

Step 6 is left as an exercise for the reader. Change the noun rule to output an indefinite article before the noun. You can do this following the instructions for adding the pronoun. The string you need to output is `^a<det><ind><sg>$`.

Don't worry if you get a `~a` in the output, this is expected, the ~ is used for postgeneration.

1-level structural transfer: conditional statements

3-level structural transfer

Resorni ministar je navlačio ljude, kaže sejte biljku zelenu i čudo će da bude

The minister of agriculture tricks the people, he says plant the green herb and there will be a miracle

Lexical transfer

```
^Resorni ministar<n><ma><sg><nom>/Minister# of agrculture<n><sg><nom>$
^biti<vbser><clt><pres><p3><sg>/be<vbser><pres><p3><sg>$
^*navlačio/trick$
^čovjek<n><ma><pl><acc>/person<n><pl><acc>$
^,<cm>/,<cm>$
^kazati<vblex><perf><tv><pres><p3><sg>/say<vblex><pres><p3><sg>$
^*sejte/plant$
^biljka<n><f><sg><acc>/herb<n><sg><acc>$
^zelen<adj><pst><ma><sg><dat><ind>/green<adj><sint><pst><ma><sg><dat><ind>$
^i<cnjcoo>/and<cnjcoo>$
^čudo<n><nt><sg><nom>/miracle<n><sg><nom>$
^htjeti<vbmod><clt><futI><p3><sg>/will<vaux><futI><p3><sg>$
^da<cnjsub>/$
^biti<vbser><futII><pres><p3><sg>/be<vbser><futII><pres><p3><sg>$
```

Notes

1. e.g. Matxin, OpenLogos, ...
2. This tag stands for *synthetic*, as in opposition to *analytic*. In English, *synthetic* adjectives inflect for comparison with *-er* and *-est*, while *analytic* ones form comparative and superlative forms with *more* and *most*.
3. You may use any apertium generator for English, one is found in `apertium/incubator/apertium-sh-en/`, and may be compiled by giving the command `sh compile.sh` in the `apertium-sh-en` catalogue. For more information, please see the Apertium New Language Pair HOWTO.

See also

- http://wiki.apertium.eu/index.php/Session_5:_Structural_transfer_basics (link dead)