Apertium New Language Pair HOWTO

En français

Apertium New Language Pair HOWTO

This HOWTO document will describe how to start a new language pair for the Apertium machine translation system from scratch. You can check the list of language pairs that have already been started.

It does not assume any knowledge of linguistics, or machine translation above the level of being able to distinguish nouns from verbs (and prepositions etc.)

Contents

Introduction

Apertium is, as you've probably realised by now, a machine translation system. Well, not quite, it's a machine translation platform. It provides an engine and toolbox that allow you to build your own machine translation systems. The only thing you need to do is write the data. The data consists, on a basic level, of three dictionaries and a few rules (to deal with word re-ordering and other grammatical stuff).

For a more detailed introduction into how it all works, there are some excellent papers on the Publications page.

You will need

- to install Apertium core, usually Install Apertium core using packaging
- install a new, empty, language pair How to bootstrap a new pair
- a text editor (or a specalised XML editor if you prefer)

This document will not describe how to install these packages, nor the new language framework. For general information please see also Installation.

What does a language pair consist of?

Apertium is a shallow-transfer type machine translation system. Thus, it basically works on dictionaries and shallow transfer rules. In operation, shallow-transfer is distinguished from deep-transfer in that it doesn't do full syntactic parsing, the rules are typically operations on groups of lexical units, rather than operations on parse trees.

At a basic level, there are three main dictionaries:

1. The morphological dictionary for language xxx: this contains the rules of how words in language xxx are inflected. In our example this will be called: `apertium-hbs.hbs.dix`
2. The morphological dictionary for language yyy: this contains the rules of how words in language yyy are inflected. In our example this will be called: `apertium-eng.eng.dix`

3. Bilingual dictionary: contains correspondences between words and symbols in the two languages. In our example this will be called: `apertium-hbs-eng.hbs-eng.dix`

In a translation pair, both languages can be either source or target for translation; these are relative terms.

There are also files for transfer rules in either direction. These are the rules that govern how words are re-ordered in sentences, e.g. *chat noir → cat black → black cat*. It also governs agreement of gender, number, etc. The rules can also be used to insert or delete lexical items, as will be described later. These files are:

- language xxx to language yyy transfer rules: this file contains rules for how language xxx will be changed into language yyy. In our example this will be: `apertium-hbs-eng.hbs-eng.t1x`
- language yyy to xxx language transfer rules: this file contains rules for how language yyy will be changed into language xxx. In our example this will be: `apertium-hbs-eng.eng-hbs.t1x`

Many of the language pairs currently available have other files, but we won't cover them here. If you want to know more about the other files, or the overall process, see Documentation, especially the Workflow diagram and Workflow reference. But the '.dix' and '.t1x'/'.t2x' files are the only ones required to generate a functional system.

Language pair

As you may have been alluded to by the file names, this HOWTO will use the example of translating Serbo-Croatian to English to explain how to create a basic system. This is not an ideal pair, since the system works better for more closely related languages. This shouldn't present a problem for the simple examples given here.

A brief note on terms

There are number of terms that will need to be understood before we continue.

The first is *lemma*. A lemma is the citation form of a word. It is the word stripped of any grammatical information. For example, the lemma of the word cats is *cat*. In English nouns, this will typically be the singular form of the word in question. For verbs, the lemma is the infinitive stripped of to, e.g., the lemma of *was* would be *be*.

The second is *symbol*. In the context of the Apertium system, symbol refers to a grammatical label. The word cats is a plural noun, therefore it will have the noun symbol and the plural symbol. In the input and output of Apertium modules these are typically given between angle brackets, as follows:

- <n>; for noun.
- <pl>; for plural.

Other examples of symbols are <sg>; singular, <p1> first person, <pri> present indicative, etc (see also: list of symbols). When written in angle brackets, the symbols may also be referred to as tags. It is worth noting that in many of the currently available language pairs the symbol definitions are acronyms or contractions of words in Catalan. For example, vbhaver — from vb (verb) and haver ("to have" in Catalan). Symbols are defined in <sdef> tags and used in <s> tags.

The third word is *paradigm*. In the context of the Apertium system, paradigm refers to an example of how a particular group of words inflects. In the morphological dictionary, lemmas (see above) are linked to paradigms that allow us to describe how a given lemma inflects without having to write out all of the endings.

An example of the utility of this is, if we wanted to store the two adjectives *happy* and *lazy*, instead of storing two lots of the same thing:

- happy, happ (y, ier, iest)
- lazy, laz (y, ier, iest)

We can simply store one, and then say "lazy, inflects like happy", or indeed "shy inflects like happy", "naughty inflects like happy", "friendly inflects like happy", etc. In this example, happy would be the paradigm, the model for how the others inflect. The precise description of how this is defined will be explained shortly. Paradigms are defined in <pardef> tags, and used in

<par> tags.

Getting started

Monolingual dictionaries

See also: Morphological dictionary, List of dictionaries, and Incubator

Let's start by making our first source language dictionary, Serbo-Croatian in our example. As mentioned above, this file will be called `apertium-hbs.hbs.dix`. The dictionary is an XML file. Fire up your text editor and type the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>

</dictionary>
```

So, the file so far defines that we want to start a dictionary. In order for it to be useful, we need to add some more entries, the first is an alphabet. This defines the set of letters that may be used in the dictionary, for Serbo-Croatian. It will look something like the following, containing all the letters of the Serbo-Croatian alphabet:

```
<alphabet>ABCČĆDDžĐEFGHIJKLLjMNNjOPRSŠTUVZŽabcčćddždđefghijkllljmnnjoprsštuvzž</alphabet>
```

Place the alphabet below the <dictionary> tag. (Note: Do not put newlines or space in alphabet!)

Next we need to define some symbols. Let's start off with the simple stuff, noun (n) in singular (sg) and plural (pl).

```
<sdefs>
   <sdef n="n"/>
   <sdef n="sg"/>
   <sdef n="pl"/>
</sdefs>
```

The symbol names do not have to be so small, in fact, they could just be written out in full, but as you'll be typing them a lot, it makes sense to abbreviate.

Unfortunately, it isn't quite so simple. Nouns in Serbo-Croatian inflect for more than just number, they are also inflected for case, and have a gender. However, we'll assume for the purposes of this example that the noun is masculine and in the nominative case (a full example may be found at the end of this document).

The next thing is to define a section for the paradigms,

```
<pardefs>

</pardefs>
```

and a dictionary section:

```
<section id="main" type="standard">

</section>
```

There are two types of sections, the first is a standard section, that contains words, enclitics, etc. The second type is an inconditional section which typically contains punctuation, and so forth. We don't have an inconditional section here.

So, our file should now look something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
   <alphabet>ABCČĆDDžĐEFGHIJKLLjMNNjOPRSŠTUVZŽabcčćddždefghijkllljmnnjoprsštuvzž</alphabet>
   <sdefs>
     <sdef n="n"/>
     <sdef n="sg"/>
     <sdef n="pl"/>
   </sdefs>
   <pardefs>

   </pardefs>
   <section id="main" type="standard">

   </section>
</dictionary>
```

Now we've got the skeleton in place, we can start by adding a noun. The noun in question will be 'gramofon' (which means 'gramophone' or 'record player').

The first thing we need to do, as we have no prior paradigms, is to define a paradigm inside the <pardefs> tag.

Remember, we're assuming masculine gender and nominative case. The singular form of the noun is 'gramofon', and the plural is 'gramofoni'. So:

```
<pardef n="gramofon__n">
   <e><p><l/><r><s n="n"/><s n="sg"/></r></p></e>
   <e><p><l>i</l><r><s n="n"/><s n="pl"/></r></p></e>
</pardef>
```

Note: the '<l/>' (equivalent to <l></l>) denotes that there is no extra material to be added to the stem for the singular.

This may seem like a rather verbose way of describing it, but there are reasons for this and it quickly becomes second nature. You're probably wondering what the <e>,

, <l> and <r> stand for. Well,

- e, is for entry.
- p, is for pair.
- l, is for left.
- r, is for right.

Why left and right? Well, the morphological dictionaries will later be compiled into finite state machines. Compiling them left to right produces analyses from words, and from right to left produces words from analyses. For example:

```
* gramofoni (left to right) gramofon<n><pl> (analysis)
* gramofon<n><pl> (right to left) gramofoni (generation)
```

Now we've defined a paradigm, we need to link it to its lemma, gramofon. We put this in the section that we've defined.

The entry to put in the <section> will look like:

```
<e lm="gramofon"><i>gramofon</i><par n="gramofon__n"/></e>
```

A quick run down on the abbreviations:

- lm, is for lemma.
- i, is for identity (the left and the right are the same).
- par, is for paradigm.

This entry states the lemma of the word, gramofon, the root, gramofon and the paradigm with which it inflects gramofon__n. The difference between the lemma and the root is that the lemma is the citation form of the word, while the root is the substring of the lemma to which suffixes are added. This will become clearer later when we show an entry where the two are different.

We're now ready to test the dictionary. Save it under `apertium-hbs.hbs.dix`, and then return to the shell. We first need to compile it (with lt-comp), then we can test it (with lt-proc). For those who are new to cygwin just take note that you need to save the dictionary file inside the home folder (for example C:\Apertium\home\Username\filename_of_dictionary). Otherwise you will not be able to compile.

```
$ lt-comp lr apertium-hbs.hbs.dix hbs-eng.automorf.bin
```

Should produce the output:

```
main@standard 12 12
```

As we are compiling it left to right, we're producing an analyser. Lets make a generator too.

```
$ lt-comp rl apertium-hbs.hbs.dix eng-hbs.autogen.bin
```

At this stage, the command should produce the same output.

We can now test these. Run lt-proc on the analyser; echo "gramofoni" (gramophones) at it, and see the output:

```
$ echo gramofoni | lt-proc hbs-eng.automorf.bin
^gramofoni/gramofon<n><pl>$
```

Now, for the English dictionary, do the same thing, but substitute the English word gramophone for gramofon, and change the plural inflection. What if you want to use the more correct word 'record player'? Well, we'll explain how to do that later.

You should now have two files:

- apertium-hbs.hbs.dix which contains a (very) basic Serbo-Croatian morphological dictionary, and
- apertium-eng.eng.dix which contains a (very) basic English morphological dictionary.

Bilingual dictionary

See also: Bilingual dictionary

So we now have two morphological dictionaries, next thing to make is the bilingual dictionary. This describes mappings between words. All dictionaries use the same format (which is specified in the DTD, dix.dtd).

Create a new file, `apertium-hbs-eng.hbs-eng.dix` and add the basic skeleton:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
   <alphabet/>
   <sdefs>
     <sdef n="n"/>
     <sdef n="sg"/>
     <sdef n="pl"/>
   </sdefs>

   <section id="main" type="standard">

   </section>
</dictionary>
```

Now we need to add an entry to translate between the two words. Something like:

```
<e><p><l>gramofon<s n="n"/></l><r>gramophone<s n="n"/></r></p></e>
```

Because there are a lot of these entries, they're typically written on one line to facilitate easier reading of the file. Again with the 'l' and 'r' right? Well, we compile it left to right to produce the Serbo-Croatian → English dictionary, and right to left to produce the English → Serbo-Croatian dictionary.

So, once this is done, run the following commands:

```
$ lt-comp lr apertium-hbs.hbs.dix hbs-eng.automorf.bin
$ lt-comp rl apertium-eng.eng.dix hbs-eng.autogen.bin

$ lt-comp lr apertium-eng.eng.dix eng-hbs.automorf.bin
$ lt-comp rl apertium-hbs.hbs.dix eng-hbs.autogen.bin

$ lt-comp lr apertium-hbs-eng.hbs-eng.dix hbs-eng.autobil.bin
$ lt-comp rl apertium-hbs-eng.hbs-eng.dix eng-hbs.autobil.bin
```

To generate the morphological analysers (automorf), the morphological generators (autogen) and the word lookups (autobil), the bil is for "bilingual".

Transfer rules

So, now we have two morphological dictionaries, and a bilingual dictionary. All that we need now is a transfer rule for nouns. Transfer rule files have their own DTD (transfer.dtd) which can be found in the Apertium package. If you need to implement a rule it is often a good idea to look in the rule files of other language pairs first. Many rules can be recycled/reused between languages. For example the one described below would be useful for any null-subject language.

Start out like all the others with a basic skeleton (`apertium-hbs-eng.hbs-eng.t1x`) :

```
<?xml version="1.0" encoding="UTF-8"?>
<transfer>

</transfer>
```

At the moment, because we're ignoring case, we just need to make a rule that takes the grammatical symbols input and outputs them again.

We first need to define categories and attributes. Categories and attributes both allow us to group grammatical symbols. Categories allow us to group symbols for the purposes of matching (for example 'n.*' is all nouns). Attributes allow us to group a set of symbols that can be chosen from. For example ('sg' and 'pl' may be grouped a an attribute 'number').

Lets add the necessary sections:

```
<section-def-cats>

</section-def-cats>
<section-def-attrs>

</section-def-attrs>
```

As we're only inflecting, nouns in singular and plural then we need to add a category for nouns, and with an attribute of number. Something like the following will suffice:

Into section-def-cats add:

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

This catches all nouns (lemmas followed by <n> then anything) and refers to them as "nom" (we'll see how that's used later).

Into the section section-def-attrs, add:

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

and then

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

The first defines the attribute nbr (number), which can be either singular (sg) or plural (pl).

The second defines the attribute a_nom (attribute noun).

Next we need to add a section for global variables:

```
<section-def-vars>

</section-def-vars>
```

These variables are used to store or transfer attributes between rules. We need only one for now so that the file can be validated,

```
<def-var n="number"/>
```

Finally, we need to add a rule, to take in the noun and then output it in the correct form. We'll need a rules section...

```
<section-rules>

</section-rules>
```

Changing the pace from the previous examples, I'll just paste this rule, then go through it, rather than the other way round.

```
<rule>
   <pattern>
     <pattern-item n="nom"/>
   </pattern>
   <action>
     <out>
       <lu>
         <clip pos="1" side="tl" part="lem"/>
         <clip pos="1" side="tl" part="a_nom"/>
         <clip pos="1" side="tl" part="nbr"/>
       </lu>
     </out>
   </action>
</rule>
```

The first tag is obvious, it defines a rule. The second tag, pattern basically says: "apply this rule, if this pattern is found". In this example the pattern consists of a single noun (defined by the category item nom). Note that patterns are matched in a longest-match first. So, say you have three rules, the first catches the pattern: `<prn> <vblex> <n>`, the second catches `<prn> <vblex>` and the third catches `<n>`. The pattern matched, and rule executed would be the first one.

For each pattern, there is an associated action, which produces an associated output, out. The output, is a lexical unit (lu).

The clip tag allows a user to select and manipulate attributes and parts of the source language (side="sl"), or target language (side="tl") lexical item.

Let's compile it and test it. Transfer rules are compiled with:

```
$ apertium-preprocess-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin
```

Which will generate a `hbs-eng.t1x.bin` file.

Now we're ready to test our machine translation system. There is one crucial part missing, the part-of-speech (PoS) tagger, but that will be explained shortly. In the meantime we can test it as is:

First, lets analyse a word, gramofoni:

```
$ echo "gramofoni" | lt-proc hbs-eng.automorf.bin
^gramofoni/gramofon<n><pl>$
```

Now, normally here the POS tagger would choose the right version based on the part of speech, but we don't have a POS tagger yet, so we can use this little gawk script (thanks to Sergio) that will just output the first item retrieved.

```
$ echo "gramofoni" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}'
^gramofon<n><pl>$
```

Now let's process that with the transfer rule:

```
$ echo "gramofoni" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin
```

It will output:

```
^gramophone<n><pl>$^@
```

- 'gramophone' is the target language (side="tl") lemma (lem) at position 1 (pos="1").
- '<n>' is the target language a_nom at position 1.
- '<pl>' is the target language attribute of number (nbr) at position 1.

Try commenting out one of these clip statements, recompiling and seeing what happens.

So, now we have the output from the transfer, the only thing that remains is to generate the target-language inflected forms. For this, we use lt-proc, but in generation (-g), not analysis mode.

```
$ echo "gramofoni" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin | \
  lt-proc -g hbs-eng.autogen.bin

gramophones\@
```

And c'est ca. You now have a machine translation system that translates a Serbo-Croatian noun into an English noun. Obviously this isn't very useful, but we'll get onto the more complex stuff soon. Oh, and don't worry about the '@' symbol, I'll explain that soon too.[1]

Think of a few other words that inflect the same as gramofon. How about adding those. We don't need to add any paradigms, just the entries in the main section of the monolingual and bilingual dictionaries.

Bring on the verbs

Ok, so we have a system that translates nouns, but that's pretty useless, we want to translate verbs too, and even whole sentences! How about we start with the verb to see. In Serbo-Croatian this is videti. Serbo-Croatian is a null-subject language, this means that it doesn't typically use personal pronouns before the conjugated form of the verb. English is not. So for example: I see in English would be translated as vidim in Serbo-Croatian.

- Vidim
- see<p1><sg>
- I see

Note: `<p1>` denotes first person

This will be important when we come to write the transfer rule for verbs. Other examples of null-subject languages include: Spanish, Romanian and Polish. This also has the effect that while we only need to add the verb in the Serbo-Croatian morphological dictionary, we need to add both the verb, and the personal pronouns in the English morphological dictionary. We'll go through both of these.

The other forms of the verb videti are: vidiš, vidi, vidimo, vidite, and vide; which correspond to: you see (singular), he sees, we see, you see (plural), and they see.

There are two forms of you see, one is plural and formal singular (vidite) and the other is singular and informal (vidiš).

We're going to try and translate the sentence: "Vidim gramofoni" into "I see gramophones". In the interests of space, we'll just add enough information to do the translation and will leave filling out the paradigms (adding the other conjugations of the verb) as an exercise to the reader.

The astute reader will have realised by this point that we can't just translate vidim gramofoni because it is not a grammatically correct sentence in Serbo-Croatian. The correct sentence would be vidim gramofone, as the noun takes the accusative case. We'll have to add that form too, no need to add the case information for now though, we just add it as another option for plural. So, in the paradigm definition just copy the 'e' block for 'i' and change the 'i' to 'e' there.

```
<pardef n="gramofon__n">
   <e><p><l/><r><s n="n"/><s n="sg"/></r></p></e>
   <e><p><l>i</l><r><s n="n"/><s n="pl"/></r></p></e>
   <e><p><l>e</l><r><s n="n"/><s n="pl"/></r></p></e>
</pardef>
```

First thing we need to do is add some more symbols. We need to first add a symbol for 'verb', which we'll call "vblex" (this means lexical verb, as opposed to modal verbs and other types). Verbs have 'person', and 'tense' along with number, so lets add a couple of those as well. We need to translate "I see", so for person we should add "p1", or 'first person', and for tense "pri", or 'present indicative'.

```
<sdef n="vblex"/>
<sdef n="p1"/>
<sdef n="pri"/>
```

After we've done this, the same with the nouns, we add a paradigm for the verb conjugation. The first line will be:

```
<pardef n="vid/eti__vblex">
```

The '/' is used to demarcate where the stems (the parts between the <l> </l> tags) are added to.

Then the inflection for first person singular:

```
<e><p><l>im</l><r>eti<s n="vblex"/><s n="pri"/><s n="p1"/><s n="sg"/></r></p></e>
```

The 'im' denotes the ending (as in 'vidim'), it is necessary to add 'eti' to the <r> section, as this will be chopped off by the definition. The rest is fairly straightforward, 'vblex' is lexical verb, 'pri' is present indicative tense, 'p1' is first person and 'sg' is singular. We can also add the plural which will be the same, except 'imo' instead of 'im' and 'pl' instead of 'sg'.

After this we need to add a lemma, paradigm mapping to the main section:

```
<e lm="videti"><i>vid</i><par n="vid/eti__vblex"/></e>
```

Note: the content of   is the root, not the lemma.

That's the work on the Serbo-Croatian dictionary done for now. Lets compile it then test it.

```
$ lt-comp lr apertium-hbs.hbs.dix hbs-eng.automorf.bin
main@standard 23 25
$ echo "vidim" | lt-proc hbs-eng.automorf.bin
^vidim/videti<vblex><pri><p1><sg>$
$ echo "vidimo" | lt-proc hbs-eng.automorf.bin
^vidimo/videti<vblex><pri><p1><pl>$
```

Ok, so now we do the same for the English dictionary (remember to add the same symbol definitions here as you added to the Serbo-Croatian one).

The paradigm is:

```
<pardef n="s/ee__vblex">
```

because the past tense is 'saw'. Now, we can do one of two things, we can add both first and second person, but they are the same form. In fact, all forms (except third person singular) of the verb 'to see' are 'see'. So instead we make one entry for 'see' and give it only the 'pri' symbol.

```
<e><p><l>ee</l><r>ee<s n="vblex"/><s n="pri"/></r></p></e>
```

and as always, an entry in the main section:

```
<e lm="see"><i>s</i><par n="s/ee__vblex"/></e>
```

Then lets save, recompile and test:

```
$ lt-comp lr apertium-eng.eng.dix eng-hbs.automorf.bin
main@standard 18 19

$ echo "see" | lt-proc eng-hbs.automorf.bin
^see/see<vblex><pri>$
```

Now for the obligatory entry in the bilingual dictionary:

```
<e><p><l>videti<s n="vblex"/></l><r>see<s n="vblex"/></r></p></e>
```

(again, don't forget to add the sdefs from earlier)

And recompile:

```
$ lt-comp lr apertium-hbs-eng.hbs-eng.dix hbs-eng.autobil.bin
main@standard 18 18
$ lt-comp rl apertium-hbs-eng.hbs-eng.dix eng-hbs.autobil.bin
main@standard 18 18
```

Now to test:

```
$ echo "vidim" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin

^see<vblex><pri><p1><sg>$^@
```

We get the analysis passed through correctly, but when we try and generate a surface form from this, we get a '#', like below:

```
$ echo "vidim" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin | \
  lt-proc -g hbs-eng.autogen.bin
#see\@
```

This '#' means that the generator cannot generate the correct lexical form because it does not contain it. Why is this?

Basically the analyses don't match, the 'see' in the dictionary is see<vblex><pri>, but the see delivered by the transfer is see<vblex><pri><p1><sg>. The Serbo-Croatian side has more information than the English side requires. You can test this by adding the missing symbols to the English dictionary, and then recompiling, and testing again.

However, a more paradigmatic way of taking care of this is by writing a rule. So, we open up the rules file (`apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin` in case you forgot).

We need to add a new category for 'verb'.

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

We also need to add attributes for tense and for person. We'll make it really simple for now, you can add p2 and p3, but I won't in order to save space.

```
<def-attr n="temps">
   <attr-item tags="pri"/>
</def-attr>

<def-attr n="pers">
   <attr-item tags="p1"/>
</def-attr>
```

We should also add an attribute for verbs.

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

Now onto the rule:

```
<rule>
   <pattern>
     <pattern-item n="vrb"/>
   </pattern>
   <action>
     <out>
       <lu>
         <clip pos="1" side="tl" part="lem"/>
         <clip pos="1" side="tl" part="a_verb"/>
         <clip pos="1" side="tl" part="temps"/>
       </lu>
     </out>
   </action>
</rule>
```

Remember when you tried commenting out the 'clip' tags in the previous rule example and they disappeared from the transfer, well, that's pretty much what we're doing here. We take in a verb with a full analysis, but only output a partial analysis (lemma + verb tag + tense tag).

So now, if we recompile that, we get:

```
$ echo "vidim" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin
^see<vblex><pri>$^@
```

and:

```
$ echo "vidim" | lt-proc hbs-eng.automorf.bin  | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin | \
  lt-proc -g hbs-eng.autogen.bin
see\@
```

Try it with 'vidimo' (we see) to see if you get the correct output.

Now try it with "vidim gramofone":

```
$ echo "vidim gramofoni" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin | \
  lt-proc -g hbs-eng.autogen.bin
see gramophones\@
```

But what about personal pronouns?

Well, that's great, but we're still missing the personal pronoun that is necessary in English. In order to add it in, we first need to edit the English morphological dictionary.

As before, the first thing to do is add the necessary symbols:

```
<sdef n="prn"/>
<sdef n="subj"/>
```

Of the two symbols, prn is pronoun, and subj is subject (as in the subject of a sentence).

Because there is no root, or 'lemma' for personal subject pronouns, we just add the pardef as follows:

```
<pardef n="prsubj__prn">
   <e><p><l>I</l><r>prpers<s n="prn"/><s n="subj"/><s n="p1"/><s n="sg"/></r></p></e>
</pardef>
```

With 'prsubj' being 'personal subject'. The rest of them (You, We etc.) are left as an exercise to the reader.

We can add an entry to the main section as follows:

```
<e lm="personal subject pronouns"><i/><par n="prsubj__prn"/></e>
```

So, save, recompile and test, and we should get something like:

```
$ echo "I" | lt-proc eng-hbs.automorf.bin
^I/PRPERS<prn><subj><p1><sg>$
```

(Note: it's in capitals because 'I' is in capitals).

Now we need to amend the 'verb' rule to output the subject personal pronoun along with the correct verb form.

First, add a category (this must be getting pretty pedestrian by now):

```
<def-cat n="prpers">
   <cat-item lemma="prpers" tags="prn.*"/>
</def-cat>
```

Now add the types of pronoun as attributes, we might as well add the 'obj' type as we're at it, although we won't need to use it for now:

```
<def-attr n="tipus_prn">
   <attr-item tags="prn.subj"/>
   <attr-item tags="prn.obj"/>
</def-attr>
```

And now to input the rule:

```
<rule>
   <pattern>
     <pattern-item n="vrb"/>
   </pattern>
   <action>
     <out>
       <lu>
         <lit v="prpers"/>
         <lit-tag v="prn"/>
         <lit-tag v="subj"/>
         <clip pos="1" side="tl" part="pers"/>
         <clip pos="1" side="tl" part="nbr"/>
       </lu>
       <b/>
       <lu>
         <clip pos="1" side="tl" part="lem"/>
         <clip pos="1" side="tl" part="a_verb"/>
         <clip pos="1" side="tl" part="temps"/>
       </lu>
     </out>
   </action>
</rule>
```

This is pretty much the same rule as before, only we made a couple of small changes.

We needed to output:

```
^prpers<prn><subj><p1><sg>$ ^see<vblex><pri>$
```

so that the generator could choose the right pronoun and the right form of the verb.

So, a quick rundown:

- `<lit>`, prints a literal string, in this case "prpers"
- `<lit-tag>`, prints a literal tag, because we can't get the tags from the verb, we add these ourself, "prn" for pronoun, and "subj" for subject.
- , prints a blank, a space.

Note that we retrieved the information for number and tense directly from the verb.

So, now if we recompile and test that again:

```
$ echo "vidim gramofone" | lt-proc hbs-eng.automorf.bin  | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin | \
  lt-proc -g hbs-eng.autogen.bin
I see gramophones
```

Which, while it isn't exactly prize-winning prose (much like this HOWTO), is a fairly accurate translation.

So tell me about the record player (Multiwords)

While gramophone is an English word, it isn't the best translation. Gramophone is typically used for the very old kind, you know with the needle instead of the stylus, and no powered amplification. A better translation would be 'record player'. Although this is more than one word, we can treat it as if it is one word by using multiword (multipalabra) constructions.

We don't need to touch the Serbo-Croatian dictionary, just the English one and the bilingual one, so open it up.

The plural of 'record player' is 'record players', so it takes the same paradigm as gramophone (gramophone__n) — in that we just add 's'. All we need to do is add a new element to the main section.

```
<e lm="record player"><i>record<b/>player</i><par n="gramophone__n"/></e>
```

The only thing different about this is the use of the tag, although this isn't entirely new as we saw it in use in the rules file.

So, recompile and test in the orthodox fashion:

```
$ echo "vidim gramofone" | lt-proc hbs-eng.automorf.bin | \
  gawk 'BEGIN{RS="$"; FS="/";}{nf=split($1,COMPONENTS,"^"); for(i = 1; i<nf; i++) printf COMPONENTS[i];
if($2 != "") printf("^%s$",$2);}' | \
  apertium-transfer apertium-hbs-eng.hbs-eng.t1x hbs-eng.t1x.bin hbs-eng.autobil.bin  | \
  lt-proc -g hbs-eng.autogen.bin
I see record players
```

Perfect. A big benefit of using multiwords is that you can translate idiomatic expressions verbatim, without having to do word-by-word translation. For example the English phrase, "at the moment" would be translated into Serbo-Croatian as "trenutno" (trenutak = *moment*, trenutno being adverb of that) — it would not be possible to translate this English phrase word-by-word into Serbo-Croatian.

Dealing with minor variation

Serbo-Croatian is an umbrella term for several standard languages, so there are differences in pronounciation and ortography. There is a cool phonetic writing system so you write how you speak. A notable example is the pronounciation of the proto-Slavic vowel *yat*. The word for dictionary can for instance be either "rječnik" (called Ijekavian), or "rečnik" (called Ekavian).

Analysis

There should be a fairly easy way of dealing with this, and there is, using paradigms again. Paradigms aren't only used for adding grammatical symbols, but they can also be used to replace any character/symbol with another. For example, here is a paradigm for accepting both "e" and "je" in the analysis. The paradigm should, as with the others go into the monolingual dictionary for Serbo-Croatian.

```
<pardef n="e_je__yat">
  <e>
    <p>
      <l>e</l>
      <r>e</r>
    </p>
  </e>
  <e>
    <p>
      <l>je</l>
      <r>e</r>
    </p>
  </e>
</pardef>
```

Then in the "main section":

```
    <e lm="rečnik"><i>r</i><par n="e_je__yat"/><i>čni</i><par n="rečni/k__n"/></e>
```

This only allows us to analyse both forms however... more work is necessary if we want to generate both forms.

Generation

Notes

1. Turns out I didn't get around to explaining it, so: The @ symbol appears because of the full stop '`.`' that is added to every translation in case it doesn't have one. For the technically minded, this is so that the tagger knows when a new sentence is starting, even if there isn't a full stop there.

See also

- Building dictionaries
- Finding errors in dictionaries
- Cookbook
- Chunking
- Contributing to an existing pair