Apertium[1]

Daniel G. Swanson
Indiana University
Bloomington, IN, 47405
dangswan@iu.edu

Abstract

This chapter presents the Apertium machine translation platform, a free and open-source rule-based system primarily focusing on providing translation systems and related language technology for otherwise underresourced languages. The history, philosophy, and technology of Apertium are presented together with extended examples.

Key Words: *polysemous words, machine translation*

1 Introduction

The Apertium machine translation platform is a free and open-source collection of largely rule-based natural language processing tools and a framework for combining them to build machine translation systems. The system is maintained by the Apertium Project, an organisation dedicated to promoting equal access to language technology for speakers of all languages, particularly in the field of translation.

The project began in 2004 as a consortium led by Mikel Forcada at the Universitat d'Alacant in Spain, with funding from the Spanish government to rewrite and extend various existing machine translation systems to produce a single, fully open-source system that would support all the official languages of Spain: Spanish, Catalan, Galician, Portuguese, and Basque (though the technology used in the original system works best with closely related languages, so the Basque-Spanish translator ended up relatively different from the rest).

Since that initial set of 5 languages, Apertium has grown to include nearly 200 languages and over 300 translation pairs, which are being worked on by over 100 developers in at least a dozen countries.

Section 2 describes the overarching philosophy of the Apertium project and how it has shaped the various architectural decisions in the project. Section 3 discusses the various

[1] **Ref:** Swanson, Daniel G. 2023. Apertium. In: Arvi Hurskainen, Kimmo Koskenniemi, and Tommi Pirinen (eds.), Rule-Based Language Technology. NEALT Monograph Series, 2:95-162. https://dspace.ut.ee/handle/10062/89595

steps of the translation pipeline, Section 4 provides some extended examples of addressing various challenges that may come up in the development process, and Section 5 concludes.

2 Philosophy

The Apertium organisation has the following mission statement:

The mission of the Apertium project is to collaboratively develop free/open-source machine translation for as many languages as possible, and in particular:

1. To give everyone free, unlimited access to the best possible machine-translation technologies.
2. To maintain a modular, documented, open platform for machine translation and other human language processing tasks
3. To favour the interchange and reuse of existing linguistic data.
4. To make integration with other free/open-source technologies easier.
5. To radically guarantee the reproducibility of machine translation and natural language processing research.

These goals have had a number of implications.

2.1 Free and Open Source

Everything created by the Apertium organisation is released free of charge under open source licenses (usually the Gnu General Public License (GPL) version 3) which allow anyone to use, modify, and build upon Apertium resources with very few restrictions.

2.2. Minimal Linguistic Theory

Apertium as a whole makes very few assumptions about how language works. It requires that text be somehow segmentable into words (at present, this segmentation is generally done on spaces though not always). These words are then represented as a dictionary headword, a part-of-speech, and a sequence of features.

Thus, for example, the phrase "The greenest cats" is represented in Apertium by
`^the<det><def><sp>$ ^green<adj><sint><sup>$ ^cat<n><pl>$,`
which can be read as "the, determiner, definite, singular or plural; green, adjective, has comparative form, superlative; cat, noun, plural"

There do exist linguistic theories which disagree with these assumptions, but the vast majority are compatible with them. Some of the specific modules used in the translation pipeline make stronger assumptions, but each of these has alternatives available and it is often possible to separate data from theory enough to be usable.

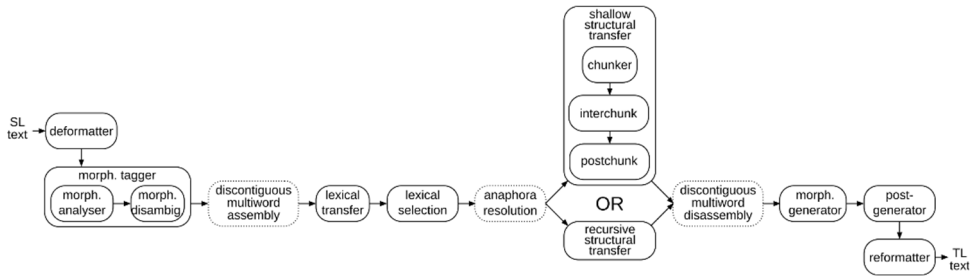

Figure 1: The sequence of modules in an Apertium pipeline. Boxes with dashed edges denote modules which included less frequently

2.3 Unix Pipelines

In order to maximise the modularity of the translation system, the entire process takes the form of a Unix pipeline, the typical steps of which are described in Section 3, where the processor for each type of rule reads from a single stream of text input in a particular format and writes to a single output stream in the same format.

This frees any individual module from having to know anything about the modules before or after it. As a result, any step of the process can be replaced by a program which operates in an entirely different way, as long as it produces output in the same format.

2.4 Fast and Deterministic Processing

Giving people access to software is not especially helpful if it requires a supercomputer to run that software. As a result, effort is made to ensure that all language data can be both compiled and used on a typical consumer laptop.

In addition, since Apertium is primarily rule-based, nearly every process will produce consistent output for a given input, ensuring that the results of any experiment can be reproduced.

3 Pipeline Structure

Apertium translation pairs typically consist of a pipeline of modules ranging from 5 to 12 steps, not counting the language-independent format handling. An outline of the typical steps in a pipeline is shown in Figure 1, and the individual modules are described below

3.1 Deformatter

The deformatter is a language-independent step which extracts any formatting information that may be present, such as when the input is a document from a word processor, rather than a plain text file. The deformatter also escapes any characters that have special meaning in the stream format.

3.2 Morphological Analyser (Monolingual Dictionary)

The morphological analyser uses a finite-state transducer to tokenise and analyse the text and convert it from surface forms to lemmas and features.

There are three common setups for compiling these transducers: lttoolbox, HFST LexC, and lexd.

The first two, lttoolbox and LexC, each fairly directly replicate the graph structure of an FST. In both of them there are "lexicons" (LexC) or "paradigms" (lttoolbox), each of which consists of a list of entries that may have some other lexicon or paradigm as a suffix (lttoolbox also supports placing a copy of a paradigm in the middle of an entry, but this is rarely used). Lexd, on the other hand, has "lexicons" which consist of entries without continuation information. The structure is instead specified in "patterns" which the compiler uses to infer the graph structure.

Examples (2), (3), and (4) show implementations in lttoolbox, LexC, and lexc, respectively, of the Hebrew nominal paradigm given in (1).

(1)	**Surface form**	**analysis**	**transliteration**	**gloss**
	ספר	ספר<n><m><sg>	sefer	book
	ספרי	ספר<n><m><pl>	sifrim	books
	בהמה	בהמה<n><f><sg>	behemah	cow
	בהמות	בהמה<n><f><pl>	behemot	cows

The vowels in the first noun change due to the placement of stress, but the following examples will only account for the consonants.

(2)

```
<dictionary>
  <pardefs>
    <pardef n="noun_f">
      <e>
        <p>
          <l>>ה/l>
          <r>ה<s n="n"/><s n="f"/><s n="sg"/></r>
        </p>
      </e>
      <e>
        <p>
          <l>>ות/l>
          <r>ה<s n="n"/><s n="f"/><s n="pl"/></r>
        </p>
      </e>
    </pardef>
    <pardef n="noun_m">
      <e>
        <p>
          <l></l>
          <r><s n="n"/><s n="m"/><s n="sg"/></r>
```

```
        </p>
      </e>
      <e>
        <p>
          <l>>י/l>
          <r><s n="n"/><s n="m"/><s n="pl"/></r>
        </p>
      </e>
    </pardef>
  </pardefs>
  <section id="main" type="standard">
    <e lm="בהמה"><i>|>בהמ/i><par n="noun_f"/></e>
    <e lm="ספר">"<i>>ספר/i><par n="noun_m"/></e>
  </section>
</dictionary>
```

Here `pardef` defines a paradigm and `section` defines the initial paradigm. Within each one, `e` is an entry, and `p` is a pair of strings, with `l` and `r` being the left and right sides of that pair, while `i` ("identity") is a pair where both sides are the same. Morphological tags are written with `s` ("symbol") and continuations to other paradigms with `par`.

For historical reasons, Apertium lttoolbox dictionaries put tags on the right, but in LexC and lexd they are on the left.

```
(3)
LEXICON Root
 NounRoot ;

LEXICON FemNoun
%<n%>%<f%>%<sg%>:ה # ;
%<n%>%<f%>%<pl%>:ות # ;

LEXICON MascNoun
%<n%>%<m%>%<sg%>: # ;
%<n%>%<m%>%<pl%>:י # ;

LEXICON NounRoot
  בהמה:|בהמ  FemNoun ;
  ספר:ספר  MascNoun ;
```

Here in the lines after each `LEXICON`, left and right strings are separated by a colon, followed by the continuation lexicon name. The `%` character is an escape and `#` is a special lexicon name referring to the final state of the FST.

```
(4)
PATTERNS
FemNounStem FemNounInfl
MascNounStem MascNounInfl

LEXICON FemNounInfl
```

```
ה:ה<n><f><sg>
ות:ה<n><f><pl>

LEXICON MascNounInfl
 <n><m><sg>:
 <n><m><pl>:י

LEXICON FemNounStem
בהמ|:בהמה

LEXICON MascNounStem
 ספר:ספר
```

Apart from the change of continuations to patterns, lexd's syntax is fairly closely based on LexC, though with fewer characters that need to be escaped.

3.3 Disambiguator

Surface forms are often ambiguous. Sometimes this is systemic, such as in Hebrew, where the second-person masculine singular is always identical to the third-person feminine singular in the imperfect. On the other hand, it can be specific to a particular word, such as between the written forms of the verb "convert" and the noun of the same spelling.

To deal with this ambiguity, the output of the morphological analyser is passed through a disambiguator. The two most common methods of doing this are using Constraint Grammar (described elsewhere in this book) or to train a Hidden Markov Model. In some languages, Constraint Grammar is applied and then the HMM handles any remaining ambiguity

3.4 Retokeniser (Discontinuous Multiword Assembly)

In the next step, the input text will be translated word-by-word, but there are often phrases that would be easier to process if they were single units. A common example of this in various languages is phrasal verbs such as English "take out" or German "an-fangen" ("start") which have multiple components that do not have to be adjacent to each other in a sentence.

To deal with this, the retokeniser uses another transducer to match sequences of words and rewrite them so that, for example, "take the trash out" becomes "take-out the trash". The transducer to do this is currently nearly always compiled from the lttoolbox format. An example entry for the English compound verb "take out" is given in (5).

(5)

```
<e>
  <p><l>take</l><r>take<b/>out</r></p>
  <i><s n="vblex"/><t/><d/></i>
  <p><l>out<s n="adv"/><d/></l><r></r></p>
</e>
```

Here most of the symbols are the same as the ones used for lexical selection, except for `t`, which indicates an arbitrary sequence of tags (since we want the compound verb to have the same inflectional information as the input), and `d`, which marks the end of a word.

3.5 Lexical Transfer (Bilingual Dictionary)

Each word in the text is then individually translated. This is done using yet another transducer, which is nearly always in lttoolbox format. The matching is done on prefixes, so an entry mapping `^cat<n>$` to `^gato<n><f>$` will map `^cat<n><sg>$` to `^gato<n><f><sg>$` and similarly for plural

3.6 Lexical Selection

Like morphological analysis, lexical transfer will often produce multiple options. There is thus need of a lexical disambiguation step analogous to the morphological one. This can be done with Constraint Grammar, and could in principle be done with a statistical model like the HMM used for morphology, though there are changes that would be needed to the HMM code that have not been implemented. In practice, however, selecting translations has nearly always been done with a module specific to that task.

The rules of this module are written in XML and compiled to a collection of transducers. An example of such rules is given in (6).

```
(6)
<rule weight="0.1">
  <match lemma="power" tags="n.*">
    <select lemma="poder" tags="n.*"/>
  </match>
</rule>
<rule weight="1.0">
  <match lemma="wind"/>
  <match lemma="power" tags="n.*">
    <select lemma="energía" tags="n.*"/>
  </match>
</rule>
```

Here the first rule indicates that the English noun “power” should be translated in general as the Spanish noun “poder”. The second rule, meanwhile, says that when the immediately preceding word in English is “wind”, then the translation should instead be “energía”. In this case the second rule will be chosen both because it has a higher weight and because its pattern includes a larger number of words

3.7 Anaphora Resolution

A few translation pairs use the anaphora resolution module to attempt to identify what prior entity in the text is being referred to, which is useful in cases where, for instance, the proper translation of a pronoun would be unclear without knowing the gender of what it referred to. The rules for this are written in XML, and an example is given in (7).

```
(7)
<!-- things that can be connected -->
<def-parameter n="detpos">
  <anaphor>
    <parameter-item has-tags="det pos"/>
  </anaphor>
  <antecedent>
    <parameter-item has-tags="n"/>
    <parameter-item has-tags="np"/>
  </antecedent>
</def-parameter>
<!-- elements in patterns -->
<def-cat n="nom">
  <cat-item has-tags="n"/>
  <cat-item has-tags="np"/>
</def-cat>
<def-cat n="com">
  <cat-item has-tags="cm"/>
</def-cat>
<!-- particular pattern -->
<markable n="AdNP">
  <pattern>
    <pattern-item n="nom"/>
    <pattern-item n="com"/>
  </pattern>
  <pattern>
    <pattern-item n="nom"/>
    <pattern-item n="nom"/>
    <pattern-item n="com"/>
  </pattern>
  <score n="-2"/>
</markable>
```

This first defines what can be a potential antecedent (thing being referred to) and what can be an anaphor (thing doing the referring). In (7), the anaphor is a possessive determiner (<det><pos>) and the antecedent is either a common noun (<n>) or a proper noun (<np>).

Then it specifies that one or two nouns immediately preceding a comma should receive a score of −2 for being the antecedent. The word with the highest total score in the 3 sentences preceding an anaphor is chosen. In this case, the score is negative because a noun followed by a comma is frequently the name or title of the person being addressed, and is likely not referred to by pronouns or determiners elsewhere in the sentence.

3.8 Structural Transfer

The purpose of the structural transfer step is to take the word-for-word translation from the bilingual dictionary and rearrange it into the proper order for the target language as well as correcting any morphological tags that differ between the two languages.

There are two modules which are used for this purpose. The first is called the "chunker" and has rules written in XML which rewrites the sequence of words a fixed number of times (typically 3). The second is referred to as "apertiumrecursive" or "recursive transfer" and has its own rule format and parses the input into a full syntax tree and rearranges it. Using this second module adds an assumption to the ones listed in Section 2.2, namely that the syntax of the source language can be usefully represented using constituency trees without movement. In practice this has not turned out to be much of a restriction for languages with relatively consistent word order, since specific rules can be written for the cases where this model doesn't hold, which tend to be either few in number or quite rare.

The rules in (8) and (9) show simple rules in chunker and recursive, respectively. Both rules transform text with adjective-noun order into noun-adjective and ensure that the noun and the adjective agree in number (singular vs plural)

```
(8)
<transfer>
  <section-def-cats>
    <def-cat n="n">
      <cat-item tags="n"/>
      <cat-item tags="n.*"/>
    </def-cat>
    <def-cat n="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="a_adj">
      <attr-item tags="adj"/>
    </def-attr>
  </section-def-attrs>
  <section-rules>
    <rule comment="adj n">
      <pattern>
        <pattern-item n="adj"/>
        <pattern-item n="n"/>
      </pattern>
      <action>
        <out>
          <lu>
            <clip pos="2" side="tl" part="whole"/>
          </lu>
          <b/>
          <lu>
            <clip pos="1" side="tl" part="lem"/>
            <clip pos="1" side="tl" part="a_adj"/>
            <clip pos="2" side="tl" part="number"/>
```

```
          </lu>
        </out>
      </action>
    </rule>
  </section-rules>
</transfer>
```

Here we first define the categories of things that can be matched by rules (`section-def-cats`), in this case nouns (`<n>`) and adjectives (`<adj>`), each with or without subsequent tags.

Next we define attributes (`section-def-attrs`), which are lists of tags that we want to be able to extract or "clip" from a particular word. Here we define `number` as consisting of singular (`<sg>`) and plural (`<pl>`). Given this definition, if we clip the `number` of a word that has a `<sg>` tag, it will evaluate to `<sg>`, but if we do the same with a word that has no number tag or that has dual number (`<du>`), then it will evaluate to an empty string.

Finally, we list the actual rules z(`section-rules`). The rule begins with a `pattern` that consists of a sequence of categories defined in `section-def-cats`. Then there is the `action`, which nearly always involves outputting something (`out`). In this case, outputting the words is the entire action, but in more complicated rules there can be conditional statements and manipulation of variables.

Within an output statement, `lu` indicates a word and `b` indicates a space. Each output word is constructed by concatenating pieces (`clip`) of the input words or of global variables. Each `clip` has a position (`pos`) specifying which input word it is using and a side specifying whether to take the tag from the source language (`sl`), the target language (`tl`), or from the antecedent selected by anaphora resolution (`ref`). They also have a `part`, which refers to one of the attributes defined in `section-def-attrs` or to one of a few special names, such as `whole` for the entire form or `lem` for the lemma.

```
(9)
number = sg pl ;
n: _.number ;
adj: _.number ;

NP: _ ;

NP -> adj n { 2 _ 1[number=2.number] } ;
```

Here we begin by defining attributes (`number = sg pl ;`) with the same meaning as in the chunker. Next we list what tags should be output for each part of speech. In this example we have declared that nouns and adjectives should have their part of speech tag (`_`) and a number tag, while a noun phrase (`NP`) should have only a part of speech tag.

Finally the rule states that an adjective and a noun can be combined into a noun phrase (`NP -> adj n`). Then, when the tree has been built, the transformation to be applied to this `NP` node is to output the second word (`2`), a space (`_`), and the first word (`1`), but with

the number tag being clipped from the second word (`2.number`) rather than from the first word.

3.9 Retokeniser (Discontiguous Multiword Disassembly)

This step is mechanically the same as the one described in Section 3.4, though it is usually splitting words apart rather than putting them together.

3.10 Morphological Generator (Monolingual Dictionary)

The morphological generator uses a transducer to convert analyses into surface forms in the target language. It is typically an inverted copy of a subset of the morphological analyser for the target language (the analyser is larger in order to account for dialectal and spelling variation which is processed but not produced).

3.11 Post-generator

The final step is the post-generator, which uses a transducer to manipulate sequences of surface forms, especially to deal with contractions in English and various Romance languages. An example for the Spanish "de" and "el" combining to give "del" is given in (10).

```
 (10)
<e>
  <p>
    <l><a/>de<b/></l>
    <r>d</r>
  </p>
  <i>el<b/></i>
</e>
```

Words that need to be manipulated by the post-generator are often generated with a tilde in front of them, so the input to this rule would be `~de el` rather than `de el`.

4 Examples

Since the challenges of morphological analysis and disambiguation are covered in more depth in the chapters on HFST and Constraint Grammar, we will focus on the later stages of the pipeline.

4.1 Separable Verbs

Consider the translation of the English "take out" to Spanish "sacar". When "take" and "out" are immediately adjacent, the rule in (5) will suffice and then we can put an entry like (11) in our bilingual dictionary.

(11)
```
<e>
  <p>
    <l>take<b/>out<s n="vblex"/></l>
    <r>sacar<s n="vblex"/></r>
  </p>
</e>
```

However, “take” and “out” don’t have to be next to each other. If the direct object is relatively short, it can go between them.

The solution to this is to adjust the rule in (5) to refer to optional words that it doesn’t modify.

```
(12)
<dictionary>
  <pardefs>
    <pardef n="maybe_noun"/>
      <e><i><w/><s n="n"/><t/><d/></i></e>
      <e></e>
    </pardef>
    <pardef n="maybe_adj"/>
      <e><i><w/><s n="adj"/><t/><d/></i></e>
      <e></e>
    </pardef>
    <pardef n="maybe_det"/>
      <e><i><w/><s n="det"/><t/><d/></i></e>
      <e></e>
    </pardef>
    <pardef n="maybe_NP">
      <e>
        <par n="maybe_det"/>
        <par n="maybe_adj"/>
        <par n="maybe_adj"/>
        <par n="maybe_adj"/>
        <par n="maybe_noun"/>
      </e>
    </pardef>
  </pardefs>
  <section id="main" type="standard">
    <e>
      <p><l>take</l><r>take<b/>out</r></p>
      <i><s n="vblex"/><t/><d/></i>
      <par n="maybe_NP"/>
      <p><l>out<s n="adv"/><d/></l><r></r></p>
    </e>
  </section>
</dictionary>
```

Here we first defined paradigms for nouns (`maybe_noun`), adjectives (`maybe_adj`), and determiners (`maybe_det`). In each one we match any string of characters with `w`, then the part of speech tag, and then any other tags with `t`. Then, since this is inside i (identity) rather than `p` (pair), we output the entire word unchanged. Then, each paradigm contains an empty entry to make itself optional.

Then we use all these to define a paradigm that matches a simple noun phrase (`maybe_NP`). The most it can match is a determiner followed by 3 adjectives and a noun, but since every preceding paradigm can also match 0 words, the noun phrase paradigm will still match if some or all of those 5 elements are not present.

Finally, we add a reference to our noun phrase paradigm in the entry for "take out". Since the paradigm can match nothing, this rule will apply to everything that the original rule applied to, but it will also apply to things that the original rule wouldn't, converting `^take<vblex><pres>$ ^the<det><def><sp>$ ^trash<n><sg>$ ^out<adv>$` into `^take out<vblex><pres>$ ^the<det><def><sp>$ ^trash<n><sg>$` where the original rule would have left it unchanged.

4.2 Adjective Order and Agreement

Consider the Zulu translations of the English phrases:

(13) English big cow
`^big<adj><sint>$ ^cow<n><sg>$`
Zulu inkomo enkulu
`^komo<n><cl9-10><sg>$ ^khulu<adj><cl9-10><sg>$`

(14) English big cows
`^big<adj><sint>$ ^cow<n><pl>$`
Zulu izinkomo ezinkulu
`^komo<n><cl9-10><pl>$ ^khulu<adj><cl9-10><pl>$`

(15) English big beautiful cow
`^big<adj><sint>$ ^beautiful<adj>$ ^cow<n><sg>$`
Zulu inkomo enhle enkulu
`^komo<n><cl9-10><sg>$ ^hle<adj><cl9-10><sg>$ ^khulu<adj><cl9-10><sg>$`

Here we need to ensure that both the noun and the adjective receive the appropriate noun class tag (`<cl9-10>`) and number tag (`<sg>` or `<pl>`) as well as changing the order so that the noun comes before the adjective.

Since the class tag on the noun is lexical information, it is best to put it in the bilingual dictionary. On the adjective, however, it is lexical information, so it should not be added, thus giving us the following pair of entries:

(16)
```
<e><p>
  <l>big<s n="adj"/><s n="sint"/></l>
  <r>khulu<s n="adj"/></r>
</p></e>
```

```
<e><p>
  <l>cow<s n="n"/></l>
  <r>komo<s n="n"/><s n="cl9-10"/></r>
</p></e>
```

Here `<sint>` is a tag marking English adjectives as having a comparative form, such as "bigger", rather than using "more".

Rules for dealing with this situation for a single adjective were presented in Section 3.8, but the examples for both systems need to be modified to handle more than one.

In the chunking system, the typical way to handle this is to define a macro to handle adjective-noun agreement such as the following:

```
(17)
<def-macro n="n-adj" npar="2">
  <let>
    <var n="adj_tags"/>
    <clip pos="2" part="tags" side="tl"/>
  </let>
  <append n="adj_tags">
    <clip pos="1" part="class" side="tl"/>
    <clip pos="1" part="number" side="tl"/>
  </append>
  <let>
    <clip pos="2" part="tags" side="tl"/>
    <var n="adj_tags"/>
  </let>
</def-macro>
```

Here we store the tags of the adjective from the dictionary in a variable named `adj_tags`, then we add the class and number tags from the noun to the end of that variable and replace the adjective's tags with the contents of the variable. If English had class or number tags on adjectives, then this could be written more simply with a `let` for each tag setting the adjective's value to the noun's value. Unfortunately for us in this instance, `let` only replaces existing tags and does nothing when the tag referred to by the first clip doesn't exist.

Now that we have a macro for agreement, we just need to write the rules for reordering, which can be done like this:

```
(18)
<rule>
  <pattern>
    <pattern-item n="adjective"/>
    <pattern-item n="noun"/>
  </pattern>
  <action>
    <call-macro n="n-adj">
      <with-param pos="2"/>
```

```
      <with-param pos="1"/>
    </call-macro>
    <out>
      <chunk name="adj_n">
        <tags>
          <tag><lit-tag v="SN"/></tag>
        </tags>
        <lu><clip pos="2" part="whole" side="tl"/></lu>
        <b/>
        <lu><clip pos="1" part="whole" side="tl"/></lu>
      </chunk>
    </out>
  </action>
</rule>
<rule>
  <pattern>
    <pattern-item n="adjective"/>
    <pattern-item n="adjective"/>
    <pattern-item n="noun"/>
  </pattern>
  <action>
    <call-macro n="n-adj">
      <with-param pos="3"/>
      <with-param pos="1"/>
    </call-macro>
    <call-macro n="n-adj">
      <with-param pos="3"/>
      <with-param pos="2"/>
    </call-macro>
    <out>
      <chunk name="adj_adj_n">
        <tags>
          <tag><lit-tag v="SN"/></tag>
        </tags>
        <lu><clip pos="3" part="whole" side="tl"/></lu>
        <b/>
        <lu><clip pos="2" part="whole" side="tl"/></lu>
        <b/>
        <lu><clip pos="1" part="whole" side="tl"/></lu>
      </chunk>
    </out>
  </action>
</rule>
```

Here the first rule matches a single adjective and then a noun while the second rule matches two. They then both apply the macro to update the tags for the adjective. Finally, each one outputs a chunk, which can then be manipulated by similar rules in the next stage of the pipeline. For the phrase in (13), the input to this rule would look like (19), below, and the output would be (20).

```
(19) ^big<adj><sint>/khulu<adj>$ ^cow<n><sg>/komo<n><cl9-10><sg>$
```

```
(20) ^adj_n<SN>{^komo<n><cl9-10><sg>$ ^khulu<adj><cl9-10><sg>$}$
```

If no rules in the interchunk or postchunk steps modify this, then the `^adj_n<SN>{}$` will be removed, leaving only the inner lemmas and tags, which is exactly what we were aiming to produce.

With recursive transfer, on the other hand, we will again have two rules, but not for the same reason.

First, we need to define the order of the tags for each part of speech.

```
(21)
n: _.class.number;
adj: _.class.number;
NP: _.class.number;
```

Here we say that nouns and adjectives will each have a part-of-speech tag followed by a noun class tag and a number tag. Unlike the previous set of rules, however, we are also putting tags on the noun phrase. This is because the easiest way to handle situations like the present one in recursive transfer involve building multiple phrase nodes and copying tags between them rather than making a single node for each quantity of adjectives.

```
(22)
NP -> %n { %1 } |
      adj %NP { %2 _ %1 } ;
```

This specifies two possible ways of assembling a noun phrase (`NP`) node. The first is to construct it from a single noun and the second is to combine an adjective with a following noun phrase constructed by a prior application of one of these rules. The part in brackets then specifies, for the corresponding node produced on the left, how to disassemble it when no more rules can be applied. Numbers refer to the elements of the input pattern (`1` being the first, `2` being the second, etc.) and underscores (`_`) indicate spaces.

In each of these rules, the percent sign (`%`) indicates that all relevant tags should be copied. On the input side they are copied from the words to the phrase and on the output side from the phrase to the words. The first rule above could be written more explicitly as in (23), but this is usually not necessary.

```
(23) NP -> n.$class.$number { 1[class=$class,number=$number] } ;
```

Here once again using the input phrase in (13), we have the input in (24), which is then built up into the tree in (25), and then disassembled into the output in (26).

```
(24) ^big<adj><sint>/khulu<adj>$ ^cow<n><sg>/komo<n><cl9-10><sg>$

(25) ^cow<NP><cl9-10><sg>{
        ^big<adj><sint>/khulu<adj>$
        ^cow<NP><cl9-10><sg>{
             ^cow<n><sg>/komo<n><cl9-10><sg>$
        }$
     }$
```

```
(26) ^komo<n><cl9-10><sg>$ ^khulu<adj><cl9-10><sg>$
```

5 Conclusion

In this chapter we have presented the Apertium machine translation platform, its history, philosophy, and goals. We have discussed how its philosophy and goals have shaped the overall pipeline and the individual modules within it. We have discussed each of the standard modules in the pipelines and given examples of their use.

References

Forcada, Mikel. L., Ginestí-Rosell, M., Nordfalk, J., O'Regan, J., Ortiz-Rojas, S., Pérez-Ortiz, J. A., Sánchez-Martínez, F., Ramírez-Sánchez, G., and Tyers, F. M. 2011:
Apertium: a free/open-source platform for rule-based machine translation. Machine translation, 25(2):127– 144. 18

Khanna, Tanmai, Washington, J. N., Tyers, F. M., Bayatlı, S., Swanson, D. G., Pirinen, T. A., Tang, I., and Alòs i Font, H. 2021:
Recent advances in apertium, a free/open-source rule-based machine translation platform for low-resource languages. Machine Translation, 35:475-502.