NSM-DALIA: A JavaScript NLP Tool for Natural Semantic Metalanguages

Francesco Zamblera

July 25, 2010

Abstract

This is `nsm-dalia`, an automatic translator for Natural Semantic Metalanguages. The code is in javascript with a HTML interface.

Code and documentation are written with Norman Ramsey's *noweb* tool for literate programming.

After a short introduction on Natural Semantic Metalanguage (1.1), the NSM-DALIA implementation of the NSM (1.2) and the particular approach to automatic translation here adopted (1.3) are briefly discussed; then the HTML interface is described (2), followed by the two main sections of the javascript code: the translation engine (3.2) and the grammar compiler (3.3).

1 Introduction

1.1 The Natural Semantic Metalanguage

The linguistic theory known as Natural Semantic Metalanguage (NSM) has been developed by Anna Wierzbicka and colleagues for some forty years.

Some basic assumptions of this theory are:

- We cannot escape from natural language(s) to describe meaning in natural language. Formalized systems used to describe natural languages need themselves natural language to be understood and interpreted (by humans). That is, natural language can function as its own metalanguage;
- To define the meaning of a word, we must use other words which are simpler than the one we want to define, otherwise we have defined nothing. Defining a word using simpler words is called *reductive paraphrasis*;
- There is a basic set of concepts, called *semantic primes*, whose meaning is undefinable; that is, there are no simpler words which we can use to paraphrase their meaning;
- These semantic primes are the same for all languages and are expressable in every language. A particular language can express a concept using a word or an affix or a syntactic construction; a word which expresses a semantic prime can also have other meanings in a language (for example, the Spanish word *querer* expresses the semantic prime WANT, but it also means *love*); however, the concepts in question can nevertheless be expressed;
- These semantic primes can be combined in some basic ways, which the theory tries to discover. These combinations are again available in every language, though the syntactic and morphological realization of these combinations differ from language to language;

- Thus, according to the NSM theory, all languages share a basic set of words and a basic set of combinatorial possibilities of these words. With these primes, each language can construct sentences and texts which are perfectly translatable in every other language. The analysis of these basic sentences gives the *grammar core* of the language.

Briefly, the NSM of a language (that is, English NSM, Spanish NSM, and so on), is a "minilanguage" which can be thought of as the basic subset of the language in question. It contains the very basic vocabulary and sentence patterns. With this "minilanguage", we can express many things, among which:

- dictionary definitions of words and expressions;
- explanations of grammatical morphemes and constructions;
- *cultural scripts*, i.e. texts which describe a basic pattern of behaviour in a given society.

1.2 NSM-DALIA

As the NSM subset of a language is both lexically and syntactically very restricted, it should be feasible to build NLP applications for NSM.

I had begun working on a PROLOG program called NSM-DALIA some years ago; this program is still being developed but it already works.

The program which is documented here is a "minor" variant of the PROLOG one, written in JavaScript. As for version 1.0, the program works essentially as an automatic translator from one NSM to another. Version 1.0 comes along with English and Spanish NSMs installed. Writing a new grammar is much easier than in PROLOG NSM-DALIA. The choice of the scripting language JavaScript allows the program to run in virtually every modern web browser, without the need to install software on the user's computer. The main drawback is that html-embedded client-side scripting languages such as JavaScript cannot read from or write to the user's disk. So the functionalities of the program are rather limited: for example, users cannot read the text to be translated from a file and write the translated output to another file, but they must open the input file in an editor, copy-and-paste the text into the input textarea of the program, and copy-and-paste the output text from the output textarea to the desired file. A Python version is being developed which has no such restriction, and comes along with other facilities, e.g. a tool to help in grammar writing.

1.3 The Grammars

Grammars are simply list of rules which are stored in a JavaScript array.

Grammars define no intermediate metalanguage; rules map syntagmas of one NSM directly to another NSM; there are also rules who perform basic morphological analysis.

Each rule has two parts:

1. a match;

2. a replacement for the match.

The translation engine is a simple cycle through the grammar array, which tries the match part of each rule against the input text. If the rule matches, the replacement is carried out.

For example, the English morphology contains rules like these:

```
["does", "3 do"],
["has", "3 have"],
```

If the input text has somewhere the verb "do", this will be analyzed into "3 do". Later, a rule for English-to-Spanish translation could replace "3 do" with "hace".

As a regular expression is equivalent to a finite state grammar, the translation engine can be thought of as an implementation of a cascade of finite state automata. Given the restrictedness of the NSM, this limited power should be sufficient. It would be a theoretically interesting result to show that Natural Semantic Metalanguages are finite-state languages, but this is beyond the scope of this paper.

1.3.1 Rules

As we have seen, *compiled* rules have two parts, a match and a replacement. The grammar author, however, can write *three-part rules*, which have the form *match_string, replace_string, variables*. Such rules are shortcuts for a list of rules with the same pattern.

Here is an example of a three-part rule:

```
["STEMn",                           //match
 "p STEM",                          //replace
 "STEM=siente,puede,mueve,piensa,\  //variables
       hace,oye,tiene,muere,vive,\
       sabe,quiere,esta'"
]
```

This rule matches, for example, a part of an input string like "piensan", and transforms it into "p piensa" (*p* stands for "third person plural").

Such a rule has the same effect of (and in fact is compiled into) a list of rules like the following:

```
["siente", "3 siente"],
["puede", "3 puede"],
["mueve", "3 mueve"],
["piensa", "3 piensa"],
...
```

And here is a three-part rule taken from the Spanish morphology section, which illustrates a more elaborate use of variables, for the analysis of the Spanish *imperfecto*:

```
["STEM1SUFF1",                                  // match
 "SUFF2 IMPF STEM2",                            // replace

 "STEM=hac|hace, mov|mueve, viv|vive, \         // variable STEM
       mor|muere, ten|tiene, sent|siente, \
       quer|quiere, v|ve, o|oye, sab|sabe, \
       pod|puede; \

  SUFF=i'an|p, i'a|3"                             // variable SUFF
]
...
```

We can call this a *complex* use of variables, as against the *simple* one seen in the previous example.

This complex rule stands for a long list of simple rules:
[“hacían”,“p IMPF hace”],
[“hacía”,“3 IMPF hace”],
[“movían”,“p IMPF mueve”],
[“movía”,“3 IMPF mueve”],
...

A complex rule is used to map values from the match to corresponding values of the replacement string. We can define a variable `v = a|b|c, aa|bb|cc`; in the match and replacement parts of the rule, we will not use simply “v”, but v_n, where n is a numerical index. v_1 will take the values a,aa; v_2 the values b,bb; and v_3 the values c,cc; and a value a for v1 will correspond to a value b for v2 and a value c for v3.

1.3.2 How Grammars Work

There are two types of grammars: L1-to-L2 translators and morphology analyzers.

A complete grammar to be used in the translation process is built by assembling a L1-to-L2 translator and two morphology grammars.

What does this mean? Let us suppose that we want to build a grammar which can allow translation between English and Spanish NSMs, in both direction. We then need write:

1. An English-To-Spanish or a Spanish-To-English translation grammar;
2. An English morphology analyzer;
3. A Spanish morphology analyzer.

In the previous section, we have seen examples of rules of morphology analyzers, but a translation grammar has the same types of rule. Here is a rule taken from the Spanish-to-English translator:

```
["<*NP*> algunas ALT1 N1 INT1 AP1",
 "<*NP*> some ALT2 INT2 AP2 N2",

   "INT=muy|very, 0|0; \
    AP=grandes|big, pequen'as|small,
       buenas|good, malas|bad, 0|0;
    ALT=otras|other,0|0;
    NP=personas|persons,cosas|things"],
```

Variables are used exactly as in the morphology examples; the only new feature of the previous rule is the initial string `<*NP*>`, meaning that when a match is found, the replaced string must be labeled as NP. In the match part, the string `<*NP*>` is simply ignored.

That is, if the translation engine finds the string "algunas cosas muy buenas", which matches the pattern, it will replace it with `<NP* some good things *>`.

Later rules can refer to this string, as well as to any string included between `<NP*` and `*>`, as simply `(*NP*)`[1].

Note also the final alternative `0|0`, meaning that the item in question is optional.

2 The HTML Interface

The javascript code works inside a web page written in HTML. The structure of the HTML file is of course:

6 ⟨*nsm-dalia-1-0.html* 6⟩≡

```
<html>
  ⟨head 7a⟩
  ⟨body 12⟩
</html>
```

[1] Note that unlike in a true context-free approach, such "phrases" cannot be nested.

2.1 The *head* tag

The `head` element contains the link to the file `nsm-dalia-<version>.js` (the javascript code), to the source code of the grammars, and to some other utility script. The `head` element also includes an initialization script, which I have placed directly in the interface:

7a ⟨*head* 7a⟩≡ (6)

```
<head>
  <title>NSM Transator (NSM-DALIA-JS version 1.0)</title>
  <meta content="">
  <style></style>

  <!-- links to the program -->
   <script type="text/javascript"
             src="grammars_js/eng_morph.js"></script>
    <script type="text/javascript"
             src="grammars_js/spaCO_eng.js"></script>
    <script type="text/javascript"
             src="nsm-dalia-1-0.js"></script>
    <script type="text/javascript"
             src="nsm-dalia-installed.js"></script>
    <script type="text/javascript"
             src="gnu-gpl.js"></script>
    <script type="text/javascript"
             src="show_gnu.js"></script>

    <script type="text/javascript">
     ⟨initialization script 7b⟩
   </script>
</head>
```

The initialization script is called automatically when the web page is loaded, using the attribute *onLoad* of the html tag *body*.

The function `newOption`, which I modified slightly from an online JavaScript tutorial (http://www.plus2net.com/javascript_tutorial/list-adding.php), creates a new option object (optn) which will be added to the select list.

7b ⟨*initialization script* 7b⟩≡ (7a) 8b ▷

```
// This function is a modified version of a function from URL:
// http://www.plus2net.com/javascript_tutorial/list-adding.php
function newOption(text,value) {
  var optn = document.createElement("option");
  optn.text = text;
  optn.value = value;
  return optn;
}
```

Defines:
`newOption`, used in chunk 8b.

The function `init_select_list` reads the javascript object `installed_lang`, which contains a list of the languages for which a grammar is available. As for version 1.0, where the languages are English and (Colombian) Spanish, the object `installed_lang` is simply:

8a ⟨*installed languages* 8a⟩≡ (13a)

```
var installed_lang = {
     eng : 'English',
     spaCO : 'Espa\u00F1ol (Colombia)'};
```

Defines:
`installed_lang`, used in chunk 8b.

After having read the list of installed languages, `init_select_list` calls the `newOption` function for each item to to insert into the select list:

8b ⟨*initialization script* 7b⟩+≡ (7a) ◁7b 8c▷

```
function init_select_list(list) {
   for (var l in installed_lang)
   list.options.add(newOption(installed_lang[l],l))
 }
```

Defines:
`init_select_list`, used in chunk 8c.
Uses `installed_lang` 8a and `newOption` 7b.

First of all, the function `init_select_lists`, the one called on the onload event, simply calls the function `init_select_list` twice, once for each list:

8c ⟨*initialization script* 7b⟩+≡ (7a) ◁8b 9a▷

```
function init_select_lists() {
   var list = document.getElementById("l1_selector");
   init_select_list(list);
   list = document.getElementById("l2_selector");
   init_select_list(list);
 }
```

Defines:
`init_select_lists`, used in chunk 12.
Uses `init_select_list` 8b.

And finally, the function `switch_intro` can be called bi clicking on the line Ĥide introduction. This function hides the banner and leaves the space available on the browser window for the text areas.

9a ⟨*initialization script* 7b⟩+≡ (7a) ◁8c

```
function switch_intro(hide) {
  var sect = document.getElementById("intro_slot");
  var switcher =  document.getElementById("switch_intro_slot");
  if (hide) {
     sect.style.display = 'none';
     switcher.innerHTML =
       '<a href="javascript:switch_intro(false);"> \
        (Show introduction)</a>';
    }
  else {
     sect.style.display = '';
     switcher.innerHTML =
         '<a href="javascript:switch_intro(true);"> \
         (Hide introduction)</a>';
    }
}
```

2.2 The *body* tag

The `body` consists of a banner which announces the program name, license, links to the GNU GPL, a short "how-to-use" paragraph, and options to hide the banner and to show the transcription table if a transcription system is needed to enter text which contains non-ascii characters (for example, the Spanish accented vowels can be entered as a' for á, i' for í, and so on).

A HTML form follows, which constitutes the interface to the javascript routines. The form contains, first of all, two textareas, one for the input of the data, the other for the script to write the output of the translation into. The two areas form the row of a table:

9b ⟨*textareas* 9b⟩≡ (12)

```
<tr>
 <td width="50%">
    <textarea rows="15" cols="50" id="t1"></textarea>
 </td>
 <td width="50%">
    <textarea rows="15" cols="50" id="t2"></textarea>
 </td>
</tr>
```

The two *id*s `t1` and `t2` will be referred to by the translator function.

Other than the text areas, the form shows the button `Translate`. A click on the button calls the function `translate()`, the motor of the translation script:

10 ⟨*translate button* 10⟩≡ (12)

```
<tr>
 <td><input type="button" value="Translate"
         onClick="translate();"></input></td>
 <td></td>
</tr>
```

Uses `translate()` 15.

The last items in the form are two select list, which contain the languages installed. The user selects L1 and L2 from these lists. The static code of the lists contains only the empty option `-- Select a grammar --` ; the actual list is built dynamically by the initialization function `init_select_lists`.

The form contains also a checkbox, which can be activated if the user cannot enter non-ascii characters directly and wants to use a transcription system. A click on the Šhow table link prints a table onto the browser window with the transcription.

11 ⟨*select languages* 11⟩≡ (12)

```
<tr>
  <td><h4 align="center">Source language</h4>
      <input type="checkbox" id="toggle_l1_tr">
          Input transcription
      </input>
      (<a href="javascript:show_l1_transcr();">
         Show table)</a>
  </td>
  <td align="center"><h4>Target language</h4></td>
</tr>

<tr>
 <td align="center">
  <select id="l1_selector" name="l1_selector"
   onChange="build_grammars(document.panel.l1_selector.value,'1');">
    <option name="xxx" value="xxx" selected="selected">
        -- Select source language --
    </option>
   </select>
  </td>

  <td align="center">
   <select id="l2_selector" name="l2_selector"
    onChange="build_grammars(document.panel.l2_selector.value,'2');">
      <option name="xxx" value="xxx" selected="selected">
          -- Select target language --
    </option>
   </select>
  </td>
</tr>
```

Uses `build_grammars` 31b 32.

So, here is the `body` of the HTML interface.

12 ⟨*body* 12⟩≡ (6)

```
<body onload="init_select_lists();">

  <!-- Title and license banner -->
  <h1 align="center">NSM Translator</h1>
  <h3 align="center">NSM-DALIA-JS, version 1.0</h3>
  <div id="intro_slot">
   <p align="center">Copyright (C) 2010 Francesco Zamblera,
      under the GNU GPL.
      (<a href="javascript:show_license()">Show the whole text
      of the GNU GPL</a>).</p>
   <p>This program comes with <strong>ABSOLUTELY NO WARRANTY</strong>;
    <a href="javascript:show_w();">click here for details</a>.
    This is free software, and you are welcome to redistribute it
    under certain conditions;
    <a href="javascript:show_c();">click here for details</a>.</p>

    <hr/>

   <!-- How-to paragraph -->
   This is a simple translator for Natural Semantic Metalanguage.
    <ol>
     <li>Choose an input language and an output language
         from the respective select boxes;</li>
     <li>Write some NSM text in the input language text area;</li>
     <li>Click on the TRANSLATE button. Translation into output
         language will appear in the respective text area.</li>
    </ol>
  </div>

  <!-- Empty slots for dynamic text -->
  <div align="center" id="switch_intro_slot">
    <a href="javascript:switch_intro(true);">
      (Hide introduction)</a></div>
  <div id="notify_slot"></div>
  <div id="license_slot"></div>
  <div id="transcr_slot"></div>

 <!-- Form -->
 <form name="panel">
  <table width="100%">
    ⟨select languages 11⟩
    ⟨textareas 9b⟩
    ⟨translate button 10⟩
  </table>
 </form>
</body>
```

Uses init_select_lists 8c.

3 The Program

The installation information is stored in the file nsm-dalia-installed.js, which contains the list of the installed languages and information for the correct building of grammars.

13a ⟨*nsm-dalia-installed.js* 13a⟩≡
⟨*installed languages* 8a⟩
⟨*paths* 33a⟩
⟨*direct and inverse grammar list* 21b⟩

3.1 Global Variables

A few global variables are needed:

- current_l1 and current_l2 contain the language code of L1 and L2 respectively, and are updated when a new language is selected from the select box;
- current_grammar is an empty array which will contain the compiled grammar;
- l1_transcr and l2_transcr are empty arrays which will contain the transcription tables for L1 and L2;
- end_of_grammar is a regex which strips the text produced by the translator of the marks <Phrase* and *>.

13b ⟨*global variables* 13b⟩≡ (33c)

```
/* Global variables */
var current_l1 ='xxx';
var current_l2 ='xxx';
var current_grammar = new Array();
var current_transcr = new Array();
var l1_transcr = [];
var l2_transcr = [];
```

⟨*end of grammar regex* 21a⟩

Defines:
current_grammar, used in chunks 15, 24a, 31b, and 32.
current_l1, used in chunks 18, 31b, and 32.
current_l2, used in chunks 31b and 32.

The functions which constitute the program can be grouped into two main sections: a *grammar compiler*, which transforms the grammar rules written by the user into a ready-to-run form, and the motor of the translation process, which takes a text written in a L1 NSM and applies the grammar rules to derive the L2 version.

3.2 The translation engine

The compiled grammar consists simply of an array of pairs. Each pair has a regular expression as its first member and a replacement string as its second member. The motor scans the array and substitutes every match for the regular expression in the input text with the replacement string.

So, the code of the translation engine is a very simple function with two parameters: a grammar and a text to translate. A *for* cycle implements the cascade of regular expressions:

14 ⟨*translate a text* 14⟩≡ (33c)

```
function apply_grammar(grammar,s) {
for (var i = 0; i < grammar.length; i++)
    s = normalize_spaces(s.replace(grammar[i][0],grammar[i][1]));
return normalize_spaces(s);
}
```

Defines:
`apply_grammar(grammar,s)`, never used.
Uses `normalize_spaces` 27b.

The function is called when the user clicks the *Translate* button. The button activates the function `translate`, which performs he following actions:

1. using the HTML DOM, it gets the input text from the first textarea;
2. as indentation is meaningful in an NSM text, the input passes through the `save_indent` function, which replaces spaces at the beginning of a line with $\#n\#$ where n is the number of spaces, because the computation process will often need to normalize multiple spaces, replacing them with a single one. The right indentation will be restored in the L2 text;
3. the translation engine is now called, and the result, after the restoration of the indentation, is written in the output textarea.

The function translate is, therefore, very simple:

15 ⟨*call the translator* 15⟩≡ (33c)

```
function translate() {
  var s = save_indent(
            detranscribe_l1(
              document.getElementById("t1").value));
  var show_res = document.getElementById("t2");
  show_res.value =
     transcribe_l2(
        restore_indent(
           apply_grammar (
              current_grammar,s)).replace(/\n\ /g,'\n'));
}
```

Defines:
`translate()`, used in chunk 10.
Uses `current_grammar` 13b, `detranscribe_l1` 17b, `restore_indent` 17b, `save_indent` 17b, and `transcribe_l2` 17b.

The two utility functions `save_indent` and `restore_indent` are as follows:

16 ⟨*utils* 16⟩≡ (33c) 17a ▷

```
function save_indent(s) {
  var spaces;

  // replace each newline with ';;'
  s = s.replace(/\r?\n\r?/g,';;');

  // replace punctuation, which could hang
  // the RegExp engine
  s = s.replace(/\?'/g,'INVQM ');
  s = s.replace(/\?/g,' QM');
  s = s.replace(/\./g,' FULLSTOP');

  // save spaces at the beinning of lines
  while (spaces = s.match(/;;(\s\s+)/) ||
         s.match(/^(\s\s+)/) ||
         s.match(/[\)\]\.](\s\s+)/))
     s = s.replace(spaces[1],
                   ' #'+spaces[1].length+'# ') ;
  return s;
}


function make_n_spaces (n) {
  // given integer n, return a string
  // containing n spaces
  var spaces = '';
  for (i=0; i<n; i++) spaces += ' ';
  return spaces
}


function restore_indent(s) {

  // restorer punctuation
  s = s.replace(/INVQM /g,'?\'');
  s = s.replace(/ QM/g,'?');
  s = s.replace(/ FULLSTOP/g, '.');

  // restore newlines
  s = s.replace(/;;/g, '\n');

  //restore spaces
  var spaces;
  while (spaces = s.match(/ #(\d+)# /))
      s = s.replace(
             spaces[0],
```

```
            make_n_spaces(
               parseFloat(spaces[1])) ;
  return s;
}
```

Uses restore_indent 17b and save_indent 17b.

The following routine transcribes the output text using the transcription table for L2. This is needed when the grammar does not use directly the orthography of the language.

17a ⟨*utils* 16⟩+≡ (33c) ◁16 17b▷

```
function transcribe_l2(s) {
  for (var i=0; i < l2_transcr.length; i++)
      s = s.replace(l2_transcr[i][0], l2_transcr[i][1]);
  return s;
}
```

Uses transcribe_l2 17b.

A parallel routine isused to “detranscribe” L1 input:

17b ⟨*utils* 16⟩+≡ (33c) ◁17a 18▷

```
function detranscribe_l1(s) {
  var t = document.getElementById("toggle_l1_tr");
  if (!t.checked)
     for (var i=0; i < l1_transcr.length; i++)
      s = s.replace(l1_transcr[i][0], l1_transcr[i][1]);
  return s;
}
```

Defines:
detranscribe_l1, used in chunk 15.
restore_indent, used in chunks 15 and 16.
save_indent, used in chunks 15 and 16.
transcribe_l2, used in chunks 15 and 17a.

The last utility functions show and hide the transcription table.

18 ⟨*utils* 16⟩+≡ (33c) ◁17b 27a▷

```
function show_l1_transcr() {
  if (current_l1 == 'xxx')
     alert('Please select a grammar first');
  else {
    var t_slot = document.getElementById("transcr_slot");

    // get transcription array
    var t = eval(current_l1 + '_transcr');

    if ((!t) || (t.length==0)) {
      t_slot.innerHTML = ' <font color="red">\
         Transcription table empty for language ' +
         current_l1 + '.</font>'
     }
    else {
      // build table
      var r1 ='<tr><td><font color="red">Transcription:</font></td>';
      var r2 ='<tr><td><font color="red">Character:</font></td>';
      for (var i=0; i < t.length; i++) {
         r1 += '<td>' + t[i][0] + '</td>';
         r2 += '<td>' + t[i][1] + '</td>';
       }
      r1 += '</tr>'; r2 += '</tr>';

      // show table
      t_slot.innerHTML = '<table border="1">' +
            r2 + '\n' + r1 + '</table>';
     }
    t_slot.innerHTML +=
       '<br/>(<a href="javascript:close_transcr_table();">\
        Hide</a>)';
  }
}

function close_transcr_table () {
  var t_slot = document.getElementById("transcr_slot");
  t_slot.innerHTML = '';
}
```

Uses current_l1 13b.

3.3 The Grammar Builder

The grammar used by the translation engine consists of the four separate pieces assembled together:

- the L1 morphology component;
- the L1 to L2 translation grammar;
- the "end of grammar" regex, which strips the syntactic markers from the generated string, and
- the L2 morphology, wihch "spells out" the output of the L1-to-L2 grammar.

The `grammar_build` function assembles the three pieces which constitute a grammar. Its parameter is the name of the variable which contains the main piece of the grammar. The name has the form *l1code_l2code*. For example, a Spanish-to English grammar is an array of rules; the name of the array is *spa_eng*. The other pieces needed to assemble the grammar are the morphologies of L1 and L2; whose names have the form *l1code_morph* and *l2code_morph*.

So, the `grammar build` function must first of all retrieve the codes of L1 and L2 from the parameter *grammar_name*:

19 ⟨*get the language codes* 19⟩≡ (20)

```
var codes = grammar_name.split('_');
var l1_code = codes[0];
var l2_code = codes[1];
```

Then, the empty array *grammar* is prepared which will contain the assembled grammar. This grammar will then be compiled.

To assemble the grammar, we first perform a check using the two functions `installed_direct` and `installed_inverse`. We must do this in order to allow for the possibility for a grammar to be used for backward translation from L2 to L1. If, for example, we want to translate from Spanish NSM to English NSM, and we have installed the grammar *spa_eng*, we will then mount a *direct* grammar; if we want to translate from English to Spanish we will mount an *inverse* one.

The last line of the function calls the routine which will compile the grammar. Here is the code:

20 ⟨*build grammar* 20⟩≡ (33c)

```
function build_grammar(grammar_name)  {
  ⟨get the language codes 19⟩
  var grammar = new Array();
  if (installed_direct(grammar_name))

    // build a direct grammar
    grammar = grammar.concat(
                eval(l1_code + '_morph'),
                eval(grammar_name+'_gr'),
                end_of_grammar,
                invert_morph(eval(l2_code + '_morph')));
  else if (installed_inverse(grammar_name))

   //build an inverse grammar
     grammar = grammar.concat(
                  eval(l1_code + '_morph'),
                  build_l2_l1(eval(l2_code+'_'+l1_code +'_gr')),
                  end_of_grammar,
                  invert_morph(eval(l2_code + '_morph')));
  if (grammar) compile_grammar(grammar)
}
```

Defines:
`build_grammar`, used in chunk 32.
Uses `build_l2_l1` 23, `compile_grammar` 24a, `end_of_grammar` 21a, `installed_direct` 21c, `installed_inverse` 21c, and `invert_morph` 22.

The global string `end_of_grammar` which is concatenated before the L2 morphology grammar is a regex which strips all the "phrase" markers from the output of the translation component[2]:

21a ⟨*end of grammar regex* 21a⟩≡ (13b)

```
var end_of_grammar = [
  ['<[^\\*]*\\*',''],
  ['\\*>','']
];
```

Defines:
`end_of_grammar`, used in chunk 20.

The functions which check whether the grammar must be built in its *direct* or in its *inverse* form only have to consult the global variables `direct_grammars` and `inverse_grammars`:

As for version 1.0, which comes with a Spanish-English grammar, these variables are simply:

21b ⟨*direct and inverse grammar list* 21b⟩≡ (13a)

```
var direct_grammars = ':spaCO_eng:';
var inverse_grammars = ':eng_spaCO:';
```

Defines:
`direct_grammars`, used in chunk 21c.
`inverse_grammars`, used in chunk 21c.

And here are the functions which consult the global variables:

21c ⟨*check installed* 21c⟩≡ (33c)

```
function installed_direct(g_name) {
   return (direct_grammars.indexOf(':' + g_name +':') != -1)
}

function installed_inverse(g_name) {
   return (inverse_grammars.indexOf(':' + g_name +':') != -1)
}
```

Defines:
`installed_direct`, used in chunks 20 and 32.
`installed_inverse`, used in chunks 20 and 32.
Uses `direct_grammars` 21b and `inverse_grammars` 21b.

[2]Phrase markers have the form $< PhraseName * ... * >$.

The morphology grammars are written for *parsing*; so an example of an English morphology rule could be:

```
["does", "3 do"],
["has", "3 have"],
["is", "3 be"],
```

that is: replace, in the input text, *does* with *3 do*, *has* with *3 have*, *is* with *3 be* and a string which consists of a *STEM + s* with *3 STEM*.

A *direct* grammar, therefore, will need the L1 morhpology in direct form, and the L2 morphology in *inverse* form. An *inverse* grammar will need the L1 morphology to be inverted, as also the main grammar. From this comes the need to distinguish direct and inverse grammars.

The segment of English morphology shown above, when inverted, becomes:

```
["3 be", "is"],
["3 do", "does"],
["3 have", "has"],
["3 be", "is"],
```

If we are generating English text from Spanish, the output of the Spanish-to-English grammar will contain strings like *3 be*, which must be converted into *is*.

The function which inverts a morphology grammar simply exchanges the "match" part with the "replace" part of each rule; furthermore, it builds the inverted grammar from the bottom up:

22 ⟨*invert morphology* 22⟩≡ (33c)

```
function invert_morph(g) {
  var new_g = new Array();
  for (var i = g.length-1; i >= 0; i--) {
      var new_rule = new Array(2);
      new_rule[0]=g[i][1];
      new_rule[1]=g[i][0];
      if (g[i][2]) new_rule.push(g[i][2]);
      new_g.push(new_rule);
    }
  return new_g;
 }
```

Defines:
`invert_morph`, used in chunk 20.

The function which inverts an L1-to-L2 grammar into an L2-to-L1 grammar is very similar; it only exchanges the "match" part with the "replace" part, but it does not alter the order of the rules:

23 ⟨*invert syntax* 23⟩≡ (33c)

```
function build_l2_l1(g) {
  var new_g = new Array();
  for (var i = 0; i < g.length; i++) {
      var new_rule = new Array(2);
      new_rule[0]=g[i][1];
      new_rule[1]=g[i][0];
      if (g[i][2]) new_rule.push(g[i][2]);
      new_g.push(new_rule);
    }
  return new_g;
 }
```

Defines:
build_l2_l1, used in chunk 20.

3.4 The Grammar Compiler

Once the grammar has been assembled, it is scanned by the function compile grammar. "Compiling" a grammar means simply replace a rule which uses variables with a list of rules without variables, in which the variables have been instantiated with their values. As for version 1.0, this is a very long process; this issue will have to be addressed in later versions.

24a ⟨*compile grammar* 24a⟩≡ (33c)

```
function compile_grammar(grammar) {

  // notify lengthy compilation
  var notify = document.getElementById("notify_slot");
  notify.innerHTML =
    '<font color="red">Compiling grammar. Please wait...</font>';
  alert('Grammar must be compiled. This will take a little time.\n\
  Please click the \"OK\" button to start compiling.');

  // compile
  for (var i = 0; i < grammar.length; i++) {
     var rule = new Array(2);
     rule[0] = grammar[i][0];
     rule[1] = grammar[i][1];
     if (grammar[i][2] && (grammar[i][2].length > 0)) {
         var vars =  grammar[i][2].split(/\s*\;\s*/);
         add_rules_with_vars(current_grammar,rule,vars);
       }
     else {  add_rule(current_grammar,rule);  }
   }

 // notify end of compilation
 var notify = document.getElementById("notify_slot");
 notify.innerHTML =
   '<font color="green">Grammar compiled successfully</font>';
}
```

Defines:
  `compile_grammar`, used in chunk 20.
Uses `add_rule` 25, `add_rules_with_vars` 28a 28b 29a 29b 29c 30a 30b, and `current_grammar` 13b.

The compile function distinguishes whether a rule has variables or not. If a rule does not have variables (i.e. `grammar[i][2]` for $rule_j$ is undefined), the rule is simply added to the compiled grammar. If the rule has variables, the `add_rule_with_vars(current_grammar,rule,vars)` routine is invoked.

24b ⟨*add rules to grammar* 24b⟩≡ (33c)
  ⟨*add compiled rule* 25⟩
  ⟨*compile rules with variables* 28a⟩

Compiling a rule without variables Before a rule without variables is added to the gramamr array, a slight modification is still to be done. The grammar writer can indicate a phrase with Phrase Name. To indicat that a noun phrase should occur in a position, the grammar writer can simply say (*NP*) in that position. This construct is turned into a regex which matches the text between the markers `<*Phrase Name` and `*>`. So, for example, (*NP*) is turned into `<NP\*\s*([^\*]*)\*>` when the gramamr is compiled.

25 ⟨*add compiled rule* 25⟩≡ (24b)

```
// adds a compiled rule to the grammar
function add_rule(grammar,rule) {
   var syntagma;
   var j=1;
   /* checks for (*SyntName*), to be replaced by a regex
        which matches the whole syntagma */
   while (syntagma = rule[0].match(/\(\*([^\*]+)\*\)/)) {
      var synt_rg = '<' + syntagma[1] +
           '\\*\\s*([^\\*]*)\\*>';
      rule[0] = rule[0].replace(
                  '(*'+syntagma[1]+'*)', synt_rg);
      rule[1] = rule[1].replace(
                  '(*'+syntagma[1]+'*)', '$' + j++);
    } //$
   rule[1] = add_phrase_marker(rule[1]);
   rule[0] = rule[0].replace(/^<\*[^\*]*\*>\s*/,'');
   rule[0] = rule[0].replace(/#/g,'\\b');
   rule[1] = rule[1].replace(/#/g,'');
   rule[0] = new RegExp(rule[0],"g");
   grammar.push(rule);
}
```

Defines:
`add_rule`, used in chunks 24a and 28a.
Uses `add_phrase_marker` 26.

On the contrary, if the grammar writer wants to give a name to a certain pattern, considering it to be a phrase, he puts `<*Phrase*>` at the beginning of the match and replace parts. For example, one of the rules which recognizes a substantive phrase in English and Spanish is:

```
["<*N*> DETN1 INT1 AP1",
 "<*N*> DETN2 INT2 AP2",
  "DETN=algo|something,alguien|someone;
   INT=muy|very, 0|0;
   AP=grande|big, pequen'o|small, bueno|good, malo|bad,
      otro|else, 0|0"],
```

the function `add_phrase_marker`, called by the previous function, would transform the previous replace pattern into `<N* DETN2 INT2 AP2 *>`, while the match part becomes simply `DETN1 INT1 AP1`.

26 ⟨*add phrase marker* 26⟩≡ (33c)

```
function add_phrase_marker(s) {
  var group  = s.match(/^<\*([^\*]*)\*>\s+/);
  if (group)
      s = '<' + group[1] + '* '
         + s.replace('<*' + group[1]
         + '*>','') + ' *>';
  return s;
}
```

Defines:
`add_phrase_marker`, used in chunk 25.

3.5 Compiling a rule with variables

The compiler for rules with variables is the longest routine in the program. It uses two utility function: a non distructive version of the array method shift, which I have called `cdr`, as the corresponding LISP command. Function `cdr` receives an array as a parameter and returns a new array which is equal to the original array without the first element, but it does not modify the original array:

27a ⟨*utils* 16⟩+≡ (33c) ◁18 27b▷

```
// non-destructive version of shift
function cdr(ar) {
 if (ar.length == 0) return [];
 else {
     var new_ar = [];
     for (var i = 1; i < ar.length; i++)
         new_ar.push(ar[i]);
     return new_ar;
   }
}
```

Defines:
`cdr`, used in chunk 29.

The second utility function replaces all multiple spaces with a single one, and trims the leading and trailing spaces:

27b ⟨*utils* 16⟩+≡ (33c) ◁27a

```
function normalize_spaces(s) {
  s = s.replace(/\s\s+/g,' ');
  s = s.replace(/^\s/,'');
  s = s.replace(/\s$/,'');
  return s;
} //$
```

Defines:
`normalize_spaces`, used in chunks 14 and 30b.

First of all, if the rule has no variables, we have reached the end of the recursion, and we pass the rule to the `add_rule` routine. Otherwile, we need to compile variables:

28a ⟨*compile rules with variables* 28a⟩≡ (24b)

```
function add_rules_with_vars(grammar,rule,vars) {
  if (vars.length == 0) // no variables
      add_rule(grammar,rule)
  else {
     ⟨compile variables 28b⟩
  }
}
```

Defines:
`add_rules_with_vars`, used in chunk 24a.
Uses `add_rule` 25.

There are two typs of variables: those who begin with a "$" followed by a digit signal that the user is using the content of variables as a regex. Every other name will indicate a string variable.

First of all, the first element of the `vars` array, `vars[0]`, is accessed. It contains the first variable group[3]. of the list of variable groups. The variable group is then split into name and content.

Then, if the variable is of the regex type, we compile a regex; if not, we compile a string variable.

28b ⟨*compile variables* 28b⟩≡ (28a)

```
      // get the list of variables as a string
      var variable_string = vars[0];

      // split the string into an array
      // whose entries are VarName = VarContent
      var variable = variable_string.split(/\s*\=\s*/);
      if (variable[0].charAt(0) == '$') {
         ⟨compile a regex variable 29a⟩
         }
      else if (variable[1] && variable[1].length>0) {
        ⟨compile a string varable 29b⟩
     } //$
```

Defines:
`add_rules_with_vars`, used in chunk 24a.

[3] With "variable group"" I mean string of the form $variable\_name = list\_of\_values$.

29a ⟨*compile a regex variable* 29a⟩≡ (28b)

```
// variables whose name are
// '$+number' indicate a regex
rule[0] = rule[0].replace(variable[0],
            '(' + variable[1] + ')');
add_rules_with_vars(
     grammar,rule,cdr(vars)); //$
```

Defines:
  `add_rules_with_vars`, used in chunk 24a.
Uses `cdr` 27a.

To compile a string variable, we first get an array of the possible values of the variable (obtained by splitting the content string at each comma). The first element in the variables array, `vars`, which contains the variable group which we are compiling now, has now been used, so we pop it[4]:

29b ⟨*compile a string varable* 29b⟩≡ (28b) 30b▷

```
var variable_content = variable[1].split(/\s*\,\s*/);
var new_vars = cdr(vars); //pop vars[0]
```

Defines:
  `add_rules_with_vars`, used in chunk 24a.
Uses `cdr` 27a.

Then, for each item in the content array, we substitute each instance of the name of the variable in the "match" and "replace" parts of the rule with that item, and call the function recursively to compile the other variables. But a preliminary check has to be made for complex variables (those which have different but corresponding values separated by a `|` character).

For simple use of variables, replacing the variable name with the content item is simply:

29c ⟨*add simple content* 29c⟩≡ (30b)

```
var variable_name = new RegExp(variable[0], "g");
new_rule[0] =
  rule[0].replace(variable_name, variable_content[j]);
new_rule[1] =
  rule[1].replace(variable_name, variable_content[j]);
```

Defines:
  `add_rules_with_vars`, used in chunk 24a.

[4] It is important not to use the JavaScript `shift` function, because it modifies the original array, which would be altered at each recursion.

For complex variables, the variable name is followed by a numerical index in the "match" and "replace" part of the rule; this index specifies which of the alternatives must be consideres a possible value for the variable.

variables with alternative values: `v=a|b|c` means v1=a, v2=b, v3=c.

30a ⟨*add alternative contents* 30a⟩≡ (30b)

```
var alternatives =
   variable_content[j].split(/\s*\|\s*/);
new_rule[0] = rule[0];
new_rule[1] = rule[1];
for (var k=0; k < alternatives.length; k++) {
   var variable_name = new RegExp(
         variable[0] + (k+1), "g");
   new_rule[0] =
      new_rule[0].replace(
        variable_name, alternatives[k]);
   new_rule[1] =
      new_rule[1].replace(
        variable_name, alternatives[k]);
}
```

Defines:
add_rules_with_vars, used in chunk 24a.

So here is the code for the string variable compilation. A *for* cycle iterates through the items of content. For each iteraton, the content item is substituted for the variable name in one of the two possible ways.

30b ⟨*compile a string varable* 29b⟩+≡ (28b) ◁29b

```
for (var j = 0; j < variable_content.length; j++) {
   var new_rule = new Array(2);
   // variables with alternative values
   if (variable_content[j].indexOf('|') != -1) {
      ⟨add alternative contents 30a⟩
    }
   else { // no alternative values
      ⟨add simple content 29c⟩
   }
   // optional variables and normalize spaces
   new_rule[0] = new_rule[0].replace(/0/g, '');
   new_rule[1] = new_rule[1].replace(/0/g, '');
   new_rule[0] = normalize_spaces(new_rule[0]);
   new_rule[1] = normalize_spaces(new_rule[1]);
   // recursion
   add_rules_with_vars(grammar,new_rule,new_vars);
 }
```

Defines:
add_rules_with_vars, used in chunk 24a.
Uses normalize_spaces 27b.

3.6 Building complex grammars

In order to translate, for example, from Spanish to French, if we have the two grammars spa_eng and fra_eng: we build a complex grammar $Spanish \rightarrow English \rightarrow French$.

31a ⟨*build complex grammars* 31a⟩≡ (33c)
⟨*build path* 33b⟩
⟨*build grammars* 31b⟩

The function `build_grammars` is the motor of the grammar compilation process. It is invoked each time the user selects a new language from one of the two select lists. The routine first assign the new language chosen to the global grammars `current_l1` or `current_l2`:

31b ⟨*build grammars* 31b⟩≡ (31a) 32▷
```
function build_grammars(code,l1_or_l2) {
  current_grammar = [];
  if (l1_or_l2 == '1')
     current_l1 = code;
  else
     current_l2 = code;
```

Defines:
`build_grammars`, used in chunk 11.
Uses `current_grammar` 13b, `current_l1` 13b, and `current_l2` 13b.

Then, if there is a choice for both L1 and L2[5], the global array `current_grammar`, which could contain a previously used grammar, is emptied. Then a check is performed: if there is a direct grammar (L1_L2) or an inverse one (L2_L1) for the language combination chosen, the `build_grammar` routine is invoked. Otherwise, a complex grammar must be built, by concatenating L1_eng and eng_L2. This is done by the `build_path` function.

32 ⟨*build grammars* 31b⟩+≡ (31a) ◁31b

```
    if ((current_l1 != 'xxx') &&
        (current_l2 != 'xxx') &&
        (current_l1 != current_l2)) {
            current_grammar = [];
            if (installed_direct(current_l1+'_'+current_l2) ||
                installed_inverse(current_l1+'_'+current_l2))
                   build_grammar(current_l1 + '_' + current_l2)
            else {
                var path = build_path(current_l1,current_l2);
                for (var i=0; i<path.length; i++)
                    build_grammar(path[i]);
              }
            build_transcr(current_l1,current_l2);
          }
   }


  function build_transcr(l1,l2) {
    var t1 = eval(l1 + '_transcr');
    var t2 = eval(l2 + '_transcr');
    l1_transcr = [];
    l2_transcr = [];
    for (var i=0; i < t1.length; i++) {
          l1_transcr[i] = new Array(2);
          l1_transcr[i][1] = t1[i][0];
          l1_transcr[i][0] = new RegExp(t1[i][1],'g');
        }
    for (var i=0; i < t2.length; i++) {
          l2_transcr[i] = new Array(2);
          l2_transcr[i][0] = new RegExp(t2[i][0],'g');
          l2_transcr[i][1] = t2[i][1];
        }
  }
```

Defines:
`build_grammars`, used in chunk 11.
`build_transcr`, never used.
Uses `build_grammar` 20, `build_path` 33b, `current_grammar` 13b, `current_l1` 13b, `current_l2` 13b, `installed_direct` 21c, and `installed_inverse` 21c.

[5] When only one is selected, the other has the dummy value 'xxx'.

The build path routine checks whether one of the two languages is English. If so, the path from L1 to L2 is already contained in one of the global variables paths_from_eng or paths_to_eng.

33a ⟨*paths* 33a⟩≡ (13a)

```
var paths_to_eng = {spaCO : ['spaCO_eng']};
var paths_from_eng = {spaCO : ['eng_spaCO']};
```

If not, a new path is built by concatenating the path from L1 to eng with the path from eng to L2.

33b ⟨*build path* 33b⟩≡ (31a)

```
function build_path(l1,l2) {
  if (l1 == 'eng')
     return paths_from_eng[l2];
  else if (l2 == 'eng')
     return paths_to_eng[l1]
  else {
      var newpath = paths_to_eng[l1];
      newpath = newpath.concat(
         paths_from_eng[l2]);
      return newpath;
    }
}
```

Defines:
`build_path`, used in chunk 32.

3.7 The Whole Program

The chunks thus defined assemble to form the whole program:

33c ⟨*nsm-dalia-1-0.js* 33c⟩≡

⟨*global variables* 13b⟩

⟨*translate a text* 14⟩
⟨*call the translator* 15⟩

⟨*utils* 16⟩

⟨*add phrase marker* 26⟩
⟨*add rules to grammar* 24b⟩
⟨*check installed* 21c⟩
⟨*invert morphology* 22⟩
⟨*invert syntax* 23⟩
⟨*compile grammar* 24a⟩
⟨*build grammar* 20⟩
⟨*build complex grammars* 31a⟩

Defined Chunks

Index