

Referent Systems: The Manual

Marcus Kracht

April 14, 2006

1 Requirements

You need to have a Unix system, with OCaml and LaTeX installed. Although there is a Windows version of OCaml, it does not offer the features we need to use OCaml. The best way around this is to first install Cygwin. This is a program that emulates a Unix platform on a Windows machine. Having installed Cygwin, you need to install OCaml and LaTeX inside Cygwin. It will *not* do to just install cygwin and operate the windows versions of the programs, since cygwin operates on its own partition and does not look at the windows partition. If you have a Mac with OS X, you should be able to use the program as is. For the most comfortable installation, it is good to have bash ("Bourne Again Shell") and "dialog". The latter is a light weight program to create graphical user interfaces. If you have neither, that is not a problem (see below).

2 Platform Issues

The program as is works both on linux and on OS X. Basically, there is no difference between the operating systems except for the system calls. If you install LaTeX on linux, it comes complete with a viewer for dvi-files, so if you have a file *name.tex*, it is enough to call (from a command line) `latex name` to process the file and then `xdvi name.dvi` to view it. In OS X, however, there are distributions that do not have a dvi-viewer; also, the commands "xdvi" and "xpdf" are often unknown. In this case, the best option is to use the sequence `pdflatex name` and then `open name.pdf`. The current version uses a check on the environment variables to determine the operating

system. It calls first `Unix.getenv "OSTYPE"`. This yields the same result as an execution of `echo $OSTYPE` in bash. This gives the result `"linux"` on linux `"Darwin"` on Mac OS X. The file `latex.ml` contains a function that performs these checks, so you need not worry about these details. It is possible to override the choices by adding explicit instructions on whether one wants to use latex or pdflatex, and what the name of the viewer is. For example, in the standalone version, there is a program `show_last ()`. If you want to specify whether you intend latex or pdflatex, add the argument `~meth:"dvi"` or `~meth:"pdf"`. If the name of the program is *name*, add the option `~view:"name"`.

3 How to produce the standalone version

The most comfortable is to run `compile`. It needs bash (Bourne Again Shell) and `"dialog"`, which is not always included. If you run the script, check the file `"compile.log"` for problems encountered during compilation. Also, in any case, it has proved useful to repeat the installation at least once (why that is I do not know). Otherwise you will get complaints that such and such file does not exist. In case you do not have `"dialog"`, you may either use

easy-compile, or execute the following sequence:

```
ocamlc -i latex.ml > latex.mli
ocamlc -a -o latex.cma -intf latex.mli latex.ml
ocamlc -i zeichen.ml > zeichen.mli
ocamlc -a -o messages.cma -intf messages.mli messages.ml
ocamlc -i messages.ml > messages.mli
ocamlc -a -o zeichen.cma -intf zeichen.mli zeichen.ml
ocamlc -i str.cma subst.ml > subst.mli
ocamlc -a -o subst.cma -intf subst.mli str.cma subst.ml
ocamlc -i dia.ml > dia.mli
ocamlc -a -o dia.cma -intf dia.mli dia.ml
ocamlc -i avs.ml > avs.mli
ocamlc -a -o avs.cma -intf avs.mli avs.ml
ocamlc -i term.ml > term.mli
ocamlc -a -o term.cma -intf term.mli term.ml
ocamlc -i sargs.ml > sargs.mli
ocamlc -a -o sargs.cma -intf sargs.mli sargs.ml
ocamlc -i parse.ml > parse.mli
ocamlc -a -o parse.cma -intf parse.mli parse.ml
ocamlc -i pavs.ml > pavs.mli
ocamlc -a -o pavs.cma -intf pavs.mli pavs.ml
ocamlc -i drs.ml > drs.mli
ocamlc -a -o drs.cma -intf drs.mli drs.ml
ocamlc -i sem.ml > sem.mli
ocamlc -a -o sem.cma -intf sem.mli sem.ml
ocamlc -i morph.ml > morph.mli
ocamlc -a -o morph.cma -intf morph.mli morph.ml
ocamlc -i entry.ml > entry.mli
ocamlc -a -o entry.cma -intf entry.mli entry.ml
ocamlc -i dct.ml > dct.mli
ocamlc -a -o dct.cma -intf dct.mli dct.ml
```

(1)

This gives you the standalone version, based only on OCaml and LaTeX. Additionally, you can create a new toplevel system as follows. (This should go either all into *one line* or else use backslash at the end of each line (this will indicate for bash that the next line continues the instruction).)

```
exec 'ocamlmktop -o referent -I ./dict unix.cma str.cma
latex.cma zeichen.cma messages.cma subst.cma dia.cma
avs.cma term.cma sargs.cma parse.cma pavs.cma drs.cma
sem.cma morph.cma entry.cma dct.cma'
```

(2)

If you have Tcl/Tk installed and want to use a graphical user interface, execute the following:

```
ocamlc -i dict-compile.ml > dict-compile.mli
ocamlc -o dc -intf dict-compile.mli str.cma unix.cma
    dict-compile.ml
./dc "en"
chmod +x rs-gui.tcl
```

(3)

(Run `./dc "de"` to get a German installation.) If you update any of the programs above, you need to go through the whole sequence. Otherwise, if you change only the dictionary, you do not need to do anything with the standalone version. For the Tk-GUI you need to repeat the first three of the four lines above:

```
ocamlc -i dict-compile.ml > dict-compile.mli
ocamlc -o dc -intf dict-compile.mli str.cma unix.cma
    dict-compile.ml
./dc "en"
```

(4)

This is because the file `rs-gui.tcl` is overwritten, so its permissions remain.

4 How to operate the program

If you have done the previous, type after the shell prompt:

```
./referent
```

(5)

This will invoke OCaml. You will then get an OCaml prompt after the “welcome” sequence. The modules are then automatically loaded. Equivalently, you may type

```
ocaml
```

 (6)

And then type

```
#use "prepare.ml";;
```

 (7)

This will load all the modules. However, involving the program **referent** will save you the extra step of loading the required modules.

Next you will have to load a dictionary. Type, for example,

```
#use "dict/deu.ml";;
```

 (8)

and the dictionary named `deu.ml` will be used. Once you have entered a dictionary, it is ready for use. Once have loaded a dictionary using the above method, it is not necessary to use the command **#use** again. Instead, you can use **load** and **cloud**. Both take as argument the name of the dictionary, without **dict/** and the suffix. So, the second time round instead of (8) you may just type

```
load "deu";;
```

 (9)

(The dictionary contains the lines `open Dct.Q;;` and `open Entry;;`, which are main level commands and can therefore not be entered into the libraries. They must be entered through the toplevel system itself. Once issued, however, it is possible to load the file via **Dynlink**, which is what the **load** command does.) Notice however that when you load a dictionary, the old dictionary is not erased. You just add more bindings. (This is useful if you want to spread a dictionary over several files.) If you want to use *another* dictionary, type

```
cloud "name";;
```

 (10)

The workspace you have consists of several things:

1. a dictionary, which in turn consists in a set of entries, and several hashtables. The dictionary can be manipulated by the following commands.
 - **refresh_lexicon ();;** adds the bindings in the hashtables. This is needed when you add an entry.
 - **show_dict ();;** to be shown the current dictionary.

- `clear_dict ()`; clears the dictionary.
- `load "name"`; loads the dictionary `dict/name.ml` on top of the existing one.
- `cload "name"`; clears the existing dictionary and loads `dict/name.ml` in its place.
- `add_entry entry`; adds the entry *entry*. See below for a description of how entries need to look like. If you add an entry that uses an identifier that is already taken, you get a warning. If you think this is in error, you might consider clearing the dictionary if you want to work with a new dictionary (or if you load a new dictionary without clearing the previous).
- `mor_add word morphlist`; adds an entry saying that *word* consists of the sequence *morpholist*. Notice that entering and entry is not enough to have it available for parsing. Even if have an entry for, say, word *dog* you must also add the line `mor_add "dog" ["dog"]`; The dictionary has otherwise no means to tell whether a unit is simple or composite.

2. a stack of entries. This stack is manipulated as follows:

- `show_last ()`; shows the topmost entry of the stack.
- `push string`; allows you to push an entry by using its identifier, which is the argument *string*.
- `pop ()`; pops the last entry from the stack
- `forward_list ()`; returns the list of sargs identified under forward multiplication of the the top two elements. (Here forward and backward mean this: let e1 be the element below the topmost element, and e2 the topmost element. Then forward multiplication consists in multiplying in the order e1 e2, with e1 the functor; backward multiplication with e2 the functor. In each case, when the multiplication attempt fails, the arguments are still lost, so beware.)
- `backward_list ()`; returns the list of sargs identified under backward multiplication of the the top two elements.
- `forward_multiply ()`; multiplies the top two elements in a forward fashion.

- `backward_multiply ()`; multiplies the top two elements in a backward fashion.
 - `diagnose dir`; where *dir* may be either "f" of "b". If it is "f" then the program returns a complete analysis of the steps involved in forward multiplying the top two elements; if it is "b" it does the same but with backward multiplication. This is useful in understanding how merge works.
 - `clear_workspace ()`; clears the stack.
3. The entire system is bilingual. Type `language ()`; and it returns the operating language. If you run "compile" you are actually given a choice as to whether to install in English or in German.
 4. The current dictionary name is stored in two languages, German and English. To see its name, type `name ()`; . To assign a new name to your dictionary, use `set_name (german name, english name)`; .
 5. `parse_show string`; parses and shows you the result. It has two optional arguments. If you omit them, the parse is for the given word order and the current language. If you add `~orders:"all"` you get parses for all permutations of the words; if you type `~language:"de"`, the result is communicated to you in German.

5 How to use and make dictionaries

Notice that dictionaries are assumed to be in the subdirectory `"/dict"` by the compilation programs (for the Tk and the html-version). The standalone version does not care about that, but it is wise to keep with this structure.

So far, dictionaries must be entered by hand. To see how they have to be made, take a look at `deu.ml`:

(A) Introductory commentary (for the user only):

```
(* Dictname=Deutsch *)
(* This is the German dictionary. *)
```

(11)

(B) To save you a lot of typing, open the modules Dct.Q and Entry:

```
open Dct.Q;;  
open Entry;;
```

(12)

(C) The dictionary names is given as a pair of strings. The first string gives the name in German (use the empty string of the same as the English string if you don't care about it), and the second string is the English name. So we define the dictionary name as follows:

```
set_name ("Deutsch","German");;
```

(13)

(D) Add entries, like this:

```
add_entry {i = "fm";  
  e = "FM";  
  a = [["x"; "duflt"; "gen=ast&cat=ob"; "gen=f&cat=ob";  
    "ptm:t:t&stm:t0:t0&rtm:t1:t1"]];  
  m = false;  
  d = "D(H()+B(L(female(x))))"};;
```

(14)

The structure is this: an entry is a record, consisting of the following fields:

- "i": the identifier; can be anything of type string.
- "e": the exponent, of type string. Basically this is the string by which one normally designates the morpheme, while the identifier is a book-keeping device.
- "a": the argument structure, given as a list of lists. The latter represent the sargs, consisting in turn of a
 1. string for the variable, which has to be a letter followed by sequence of digits
 2. a string for the diacritics,
 3. a string to define the in-avs,
 4. a string to define the out-avs,

5. a string to define the pavs.

- "m": the morphology of type boolean;
- "d": the semantics, of type string.

Any number of entries can be entered.

The string for the diacritics is composed from several letters, which may be given in any order. These are

- d if the AIS is a functor,
- u if the AIS is an argument,
- l if the variable is identified to the left,
- r if the variable is identified to the right,
- f if the merge operation to be used is fusion.

The string for the in-avs is a string of the following form. A pair attribute+values is coded as a string *att=value1+value2+...+valuen*. A sequence of pairs is written as follows: *pair1&pair2&...&pairn*. out-avs can be left empty if it is identical with the in-avs.

The pavss are entered as follows. A single statement is written in the form *att:varin:varout*, where *att* is the role predicate, *varin* is the input name and *varout* the output name. These names can be left empty. (Note that the empty sequence is not a variable name, but is treated as the absence of a variable name.) A sequence of such statements is entered with *&* as a separator.

The semantics is defined as follows. If you take a look at *drs.ml*, you will find that the type of a DRS is recursive, and uses a number of other types. Literals have the form *pred(term1,..., termn)*. Notice that we do not distinguish terms from atomic formulae. A DRS consists of a head and a body. The head is a set of variables, the body a list of clauses. A clause is either a DRS, an equation, a literal, a unary constructor applied to a DRS, a binary constructor applied to two DRSs, a unary quantifier applied to some variables and a DRS, or a binary quantifier applied to some variables and two DRSs.

1. $H(L)$: L a sequence of strings, separated by + gives a head;

2. $B(L)$: L a list with $+$ as separator, defines a body;
3. $L(P)$: P a list formed by using $+$. The first member is a predicate, the others are arguments. L stands for "Literal".
4. $E(T+U)$: T a term, U a term gives an equation;
5. $D(H+B)$: for DRS consisting of head H and body B ;
6. $U(S+D)$ for a unary constructor S and a DRS D ;
7. $P(S+H+D)$ for a unary quantifier S , a head H and a DRS D ;
8. $Q(S+H+D1+D2)$ for the quantified DRS with quantifier name Q , variable set H , nucleus $D1$ and scope $D2$;
9. $J(S+D1+D2)$ for a binary constructor S and DRSs $D1$ and $D2$;
10. $T(s)$ for a string s defines a term.

There is a special unary operator "null" such that `UnDrs("null", d)` is a clause, where `d` is a DRS. There are no sanity checks on the DRS constructor names.

(E) Add morphological decompositions:

```
mor_add "Lehrer" ["2"; "mSc"; "s"; "n"];;  
mor_add "Lehrer" ["2"; "mSc"; "s"; "g"];;
```

(15)

A morphological decomposition consists in a string (the word you type in eventually, when you parse) and a sequence of morphemes. The morphemes must appear in the list under (D) or else the parse will fail for that word. You can add any number of such bindings of words to sequences of morphemes.

(F) Create the hashtables:

```
refresh_lexicon ();;
```

(16)

(G) That's all.