

# Referent Systems 5.0: The Manual

Marcus Kracht

December 11, 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 for bash. If you have neither, that is not a problem (see below).

For best benefit, it is good to know something about Unix systems. Ocaml, if you have installed it, comes with a manual that you may consult if there are problems. The software should give no such (I hope).

## 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 `options.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 install and compile the system

First, you need to get `referent_v5.tar` (called *archive*). Then create a directory *RefSys*, move the archive there, go there and unpack the archive:

```
mkdir RefSys
mv referent_v5.tar RefSys
cd RefSys
tar xvf referent_v5.tar
```

(1)

Also, you should add the absolute path to *RefSys* to your profile so that you can execute the binaries without issuing any prefixes. Otherwise, instead of a simply issuing *command* you need to type

```
absolute-path-to-RefSys/bin/command
```

(2)

This creates subdirectories `bin` (where all executables go), `dict` (where the dictionaries go), `manuals` (for the manuals) and `lib` (for the Ocaml libraries). The system installation thus is not mingled with any of your own. The next step is to install the system. For that you do *not* need `make` (an installation tool).

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 use `od-dialog`. If you just want to do a quick install, use `easy-compile`. You may view these scripts to see which commands to use for individual compilation of the modules.

Both scripts, `easy-compile` and `od-dialog`, are interactive and give you a variety of choices. They first ask you in which language you want to perform the installation. This will determine both the installation language (currently English and German) as well as the language in which the system interfaces and output files are generated. The latter options can be overridden.

The next stage is the compilation of the system, which you may skip if you have made no changes and have a compiled system. The third step is the compilation of the dictionaries. The same applies to this step. If you have Tcl/Tk installed you may also create a graphical user interface.

## 4 How to operate the program

If you have done the previous and opted for a standalone version, you should find in the directory `bin` a file called `referent`. Type `ls -al bin/referent` and you should be able to see that it exists and is executable. Likewise, if you have opted for a Tk-user interface, you may use `ls -al bin/rs` to see that you have an executable file named `rs`. I shall not describe in detail the workings of the Tk-user interface. You can invoke it by typing `rs`.

Now type after the shell prompt:

```
referent
```

(3)

This will invoke OCaml. You will then get an OCaml prompt after the "welcome" sequence. The modules are then automatically loaded.

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

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

(4)

and the dictionary named `deu.ml` will be used. Important note: commands must always be ended by a double semicolon, which I will not use henceforth.

Once you have entered a dictionary, it is ready for use. Once you have loaded a dictionary using the above method, it is not necessary to use the

command `#use` again. Instead, you can use `load` and `cload`. 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";;
```

(5)

(Remember the double semicolon! The dictionary contains the lines `open Standalone.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

```
cload "name";;
```

(6)

**Important Notice:** You can always type `help ();;` to bring up this manual.

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.
  - `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).

- `word_add` *word* adds a word to the wordlist. The latter is simply a lookup from strings to entries, used in creating the interfaces. Even if have an entry for, say, word *dog* you must also add the line `word_add "dog"` to see it in the TCl/Tk interface (even though to enter it there you need not have it explicitly listed).
  - `by_identifier` *string* allows you to get an entry from the dictionary by its identifier rather than its surface string. This is useful if you want to check you dictionaries.
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
  - `merge_list` `()` returns the list of sargs identified under merge of the the top two elements. If merge fails, the elements are returned to the stack; otherwise they disappear and the result is put on the top instead.
  - `merge` `()` merge the top two elements. The order is always (1) functor (2) argument irrespective of the surface order in which they appear. A different is only when you use evaluated, where you may use forward and backward conventions.
  - `diagnose` `()` This returns a complete analysis of the steps. This is useful in understanding how merge works.
  - `clear_workspace` `()` clears the stack.
  - `evaluate` *string* evaluates a string consisting of identifiers and elements of the form `<` and `>`, depending on whether you want to forward multiply or backward multiply (that is, whether you use `merge e1 e2` or `merge e2 e1`). You may use brackets `(` and `)` for your convenience but they are ignored (and therefore not checked for consistency).
3. The entire system is multilingual. 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 type `~language:"de"`, the result is communicated to you in German.
6. `show_parse ()` Show you the parse of the last item in detail. Only look at it when you are really interested.

## 5 System Options

When the system is set up, a number of options are recorded and set. You find in the files `bin/VARS.LGE` and `bin/VARS.SYS` information about the language and the operating system (on some installations of OCaml there was no way to call `uname` from inside Ocaml, so I use a file to have the option ready). They are set by the installer script. When the system is compiled, the values for other options are set. You find them listed also in `options.ml`:

- `osys`: the operating system
- `meth`: the viewing method (dvi or pdf)
- `view`: the command to call the viewer
- `col`: the column size (used for latex to determine how wide to format the tables; default is 38)
- `farbe`: whether or not to use colours in output (set to "true")
- `lang`: the output language (in ISO 639)
- `transsem`: whether or not semantics is discarded when applying transformers (default "true")
- `det`: whether or not terms are issued with all list details (default "false")
- `ddet`: the way to render handlers (default "num")

- `edet`: the way to render exponents (default "dots", that means that the morpheme boundaries are show with a colon)

Each of te options can be set individually. Here are the commands:

- `sys ()` Queries the system; cannot be reset, naturally.
- `meth ()` Queries the method; reset with `setmeth method`.
- `view ()` Queries the viewer command; can be rest with `setview command`
- `det ()`; reset with `setdet bool`
- `edet ()`; reset with `setedet string`
- `ddet ()`; reset with `setddet string`
- `language ()`; reset with `setlanguage string`
- `colwidth ()`; reset with `setcolwidth int`
- `colour ()`; reset with `setcolour bool`
- `trans ()`; reset with `settrans bool`

## 6 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. *)
```

(7)

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

```
open Standalone.Q;;
open Entry;;
```

(8)

(C) The dictionary names is given as a set of pairs of strings. The first string states the name of the language (using ISO 639 two letter codes) and the second is a string that you choose to identify the dictionary *in that language* (mostly English, I presume, so the first member will be "en"). So we define the dictionary name as follows:

```
add_name ("en","German");;
add_name ("de","German");;
add_name ("hu","Nmet");;
```

(9)

There is no limit on the languages you use (or the codes), though for most of them no use can be made for lack of messages and headers (see below). Additionally, a dictionary contains the name of a *locale*. The locale is needed in sorting the dictionaries (recall that other languages have different characters and different ways of ordering words). You may freely set the locale by issuing:

```
set_locale "hu";;
```

(10)

There are standard references for locales, so I will not repeat anything here.

(D) Add entries, like this:

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

(11)

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

- "i": the identifier; can be anything of type string.
- "m": the morpheme. Its structure is looked at further below.
- "a": the argument structure, given as a list of lists. The latter represent the sargs, consisting in turn of
  1. a 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.
- "s": 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,
- t if the AIS is a transformer,
- 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.

*Notice:* When using arbitrary strings, it is wise to avoid including delimiters of any sort. These are: (, ), +, &, =, :, and +. It is best to avoid blanks too, even though that should not create problems. The best practice is to use plain letters (upper and lowercase) and numbers. Underscore is also fine, as are the remaining punctuation marks. If you understand the syntax of the expressions well you may find that symbols that are not used in delimiting a given expression type can then be included into your string. For example, the strings *att* and *value* may contain :, since it is not used as a delimiter in AVSs.

The morphemes are sets of morphs. This said, they are issued in the following way: a morpheme is given as a *list*, each of which has the following structure:

- "id": the identifier; can be anything of type string.
- "ex": the exponent; a list of strings. Internally, this is translated into an array of strings, the *exponent*.

- "ar": the argument structure, given as a list of records. The records consist of the following field labels: of
  1. f, *function*, a string, representing a simplified function;
  2. h, *handler*, a list of lists, containing pairs of numbers and booleans; if the list has  $n$  members, then each member characterises one of the members of the resulting output exponent (recall that the output exponent is composed of several strings). In turn, the  $i$ th member is a list, specifying each of the parts that constitute it; the parts may come from the functor or the argument. If it is the  $j$ th part of the functor, the item must be  $(j, \text{true})$ , and otherwise  $(j, \text{false})$ .
  3. mi, the *input morphology*. input like an AVS structure. This is described in detail; there are some exceptional additions to the field mo.
  4. mo, the *output morphology*, consisting of a string specifying an AVS plus an extra `dia=t`, `dia=` or `dia=a` and an extra `contact=lc` (to signal that the last character of the argument must be `c`) `contact=rc` (to signal that the first character of the argument must be `c`) `contact=Lc` (to signal that the last character of the argument must be anything but `c`) `contact=lc` (to signal that the first character of the argument must be anything but `c`). (Evidently, this must be redone in a nicer format.)

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  $\text{UnDrs}(\text{"null"}, d)$  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"];;
```

(12)

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.

**(G)** That's all.