HDRUG Reference Manual

Table of Contents

1. HDRUG: A Development Environment for Logic Grammars

Hdrug is an environment to develop grammars, parsers and generators for natural languages. The system provides a number of visualisation tools, including visualisation of feature structures, syntax trees, type hierarchies, lexical hierarchies, feature structure trees, definite clause definitions, grammar rules, lexical entries, chart datastructures and graphs of statistical information e.g. concerning cputime requirements of different parsers. Visualisation can be requested for various output formats, including ASCII text format, TK Canvas widget, LaTeX output, DOT output, and CLiG output.

Extendibility and flexibility have been major concerns in the design of Hdrug. The Hdrug system provides a small core system with a large library of auxiliary relations which can be included upon demand. Hdrug extends a given NLP system with a command interpreter, a graphical user interface and a number of visualisation tools. Applications using Hdrug typically add new features on top of the functionality provided by Hdrug. The system is easily extendible because of the use of the Tcl/Tk scripting language, and the availability of a large set of libraries. Flexibility is obtained by a large number of global flags which can be altered easily to change aspects of the system. Furthermore, a number of hook predicates can be defined to adapt the system to the needs of a particular application.

The flexibility is illustrated by the fact that Hdrug has been used both for the development of grammars and parsers for practical systems but also as a tool to experiment with new theoretical notions and alternative processing strategies. Furthermore, Hdrug has been used extensively both for batch processing of large text corpora, and also for demonstrating particular applications for audiences of non-experts.

Hdrug is implemented in SICStus Prolog version 3, exploiting the built-in Tcl/Tk library. The Hdrug sources are available free of charge under the Gnu Public Licence copyright restrictions.

1.1 Interface

Hdrug provides three ways of interacting with the underlying NLP system:

- Using an extendible command interpreter.
- Using Prolog queries.
- Using an extendible graphical user interface (based on Tcl/Tk).

The first two approaches are mutually exclusive: if the command interpreter is listening, then you cannot give ordinary Prolog commands and vice versa. In contrast, the graphical user interface (with mouse-driven menu’s and buttons) can always be used. This feature is very important and sets Hdrug apart from competing systems. It implies that we can use at the same time the full power of the Prolog prompt (including tracing) and the graphical user

interface. Using the command interpreter (with a history and alias mechanism) can be useful for experienced users, as it might be somewhat faster than using the mouse (but note that many menu options can be selected using accelerators). Furthermore, it is useful for situations in which the graphical user interface is not available (e.g. in the absence of an X workstation). The availability of a command-line interface in combination with mouse-driven menu's and buttons illustrates the **flexible** nature of the interface.

An important and interesting property of both the command interpreter and the graphical user interface is **extendibility**. It is very easy to add further commands (and associated actions) to the command interpreter (using straightforward DCG syntax). The graphical user interface can be extended by writing Tcl/Tk scripts, possibly in combination with some Prolog code. A number of examples will be given in the remainder of this paper.

Finally note that it is also possible to run Hdrug without the graphical user interface present (simply give the **notk** option at startup). This is sometimes useful if no X workstation is available (e.g. if you connect to the system over a slow serial line), but also for batch processing. At any point you can start or stop the graphical user interface by issuing a simple command.

1.2 Visualisation

Hdrug supports the visualisation of a large collection of data-structures into a number of different formats.

These formats include (at the moment not all datastructures are supported for all formats. For example, plots of two dimensional data is only available for Tk):

- ASCII art
- Tk Canvas
- LaTeX
- CLiG
- DOT

The data-structures for which visualisation is provided are:

- Trees. Various tree definitions can exist in parallel. For example, the system supports the printing of syntax trees, derivation trees, type hierarchy trees, lexical hierarchies etc. Actions can be defined which are executed upon clicking on a node of a tree. New tree definitions can be added to the system by simple declarations.
- Feature structures. Clicking on attributes of a feature-structure implode or explode the value of that attribute. Such feature structures can be the feature structures associated with grammar rules, lexical entries, macro definitions and parse results.

- Trees with feature structure nodes. Again, new tree definitions can be declared. An example is `http://www.let.rug.nl/~vannoord/Hdrug/Manual/dt.png`

- Graph (plots of two variable data), e.g. to display the (average) cputime or memory requirements of different parsers.

- Tables.

- Prolog clauses.

- Definite clauses with feature structure arguments. This can be used e.g. to visualise macro definitions, lexical entries, and grammar rules (possibly with associated constraints).

1.3 Parser and Generator Management

Hdrug provides an interface for the definition of parsers and generators. Hdrug manages the results of a parse or generation request. You can inspect these results later. Multiple parsers and generators can co-exist. You can compare some of these parsers with respect to speed and memory usage on a single example sentence, or on sets of pre-defined example sentences. Furthermore, actions can be defined which are executed right before parsing (generation) starts, or right after the construction of each parse result (generation result), or right after parsing is completed.

1.4 Useful Libraries

Most of the visualisation tools are available through libraries as well. In addition, the Hdrug library contains mechanisms to translate Prolog terms into feature structures and vice versa (on the basis of a number of declarations). Furthermore, a library is provided for the creation of ‘Mellish’ Prolog terms on the basis of boolean expressions over finite domains. The reverse translation is provided too. Such terms can be used as values of feature structures to implement a limited form of disjunction and negation by unification.

A number of smaller utilities is provided in the library as well, such as the management of global variables, and an extendible on-line help system.

2. Hdrug Applications

This chapter shortly lists a number of example applications which are part of the Hdrug distribution.

2.1. Ale

The Attribute-Logic Engine by Bob Carpenter and Gerald Penn is a freeware logic programming and grammar parsing and generation system. The following description is quote from the Ale Homepage
`http://www.sfs.nphil.uni-tuebingen.de/~gpenn/ale.html`:

> [Ale] integrates phrase structure parsing, semantic-head-driven generation and constraint logic programming with typed feature structures as terms. This generalizes both the feature structures of PATR-II and the terms of Prolog II to allow type inheritance and appropriateness specifications for features and values. Arbitrary constraints may be attached to types, and types may be declared as having extensional structural identity conditions. Grammars may also interleave unification steps with logic program goal calls (as can be done in DCGs), thus allowing parsing to be interleaved with other system components. ALE was developed with an eye toward Head-Driven Phrase Structure Grammar (HPSG), but it can also execute PATR-II grammars, definite clause grammars (DCGs), Prolog, Prolog-II, and LOGIN programs, etc. With suitable coding, it can also execute several aspects of Lexical-Functional Grammar (LFG).

2.2. Alvey NL Tools

Definite-clause version of the grammar of the Alvey NL Tools, and a fragment of the lexicon. Thanks to John Carroll for making the grammar and test-set available. The accompanying README file states:

NOTICE: these files were supplied by John Carroll, `johnca@cogs.susx.ac.uk`, and are derived from a version of the ALVEY Natural Language Tools. Current information about these ALVEY NL Tools is available at
`http://www.cl.cam.ac.uk/Research/NL/anlt.html`

2.3. CFG

Tiny context-free grammar. Illustration what you need to do minimally to adapt a grammar / parser to Hdrug.

2.4. Constraint-based Categorial Grammar

Constraint-based Categorial Grammar for Dutch written by G.Bouma, slightly adopted by G. van Noord for Hdrug.

2.5. Definite Clause Grammar

Tiny DCG. Illustration what you need to do minimally to adapt a grammar / parser to Hdrug.

2.6. Chat-80

The classic Chat-80 system by Fernando Pereira and David Warren

2.7. Tree Adjoining Grammar

Small Tree Adjoining Grammar with nine (related) head-corner parsing algorithms for **headed** Lexicalized and Feature-based TAG's.

2.8. Semantic-head-driven Generation and Head-corner Parsing

DCG for Dutch, originally used as illustration for semantic-head-driven generation. Furthermore, some of the parsers were used for the timings of the paper co-authored with G. Bouma on the potential efficiency of head-driven parsing.

2.9. Extraposition Grammar

Extraposition grammars as described in Pereira's CL paper.

2.10. Delayed Evaluation of Lexical Rules

HPSG grammar for Dutch using delayed evaluation techniques to implement recursive lexical rules.

2.11. Stochastic Definite Clause Grammar

Experimental Stochastic Definite Clause Grammar by Robert Malouf.

2.12. Stochastic Head-driven Phrase Structure Grammar

Experimental Stochastic Head-driven Phrase Structure Grammar by Robert Malouf. This material was used by Rob's ESSLLI course (with Miles Osborne) in Helsinki 2001.

3. Command Interpreter

In principle there are three ways to interact with Hdrug:

- SICStus Prolog Top-level
- Command Interpreter
- Graphical User Interface

The first two items are mutually exclusive: if the command interpreter is listening, then you cannot give ordinary SICStus Prolog commands and vice versa. The graphical user interface can be used in combination with both the SICStus Prolog Top-level or the command interpreter.

Prolog queries are given as ordinary Sicstus commands. This way of interacting with Hdrug can be useful for low level debugging etc. Using the command interpreter can be useful for experienced users, as it might be somewhat faster than using the graphical user interface.

The command-interpreter features a history and an alias mechanism. It includes a facility to escape to Unix, and is easily extendible by an application.

The command interpreter is started by the Prolog predicate r/0 The command interpreter command **p** halts the command interpreter (but Hdrug continues).

Commands for the command interpreter always constitute one line of user input. Such a line of input is tokenized into a number of *words using spaces and tabs as separation symbols. The first **word** is taken as the command; optional further words are taken as arguments to the command. Each command will define certain restrictions on the number and type of arguments it accepts.

Each word is treated as a Prolog atomic (either atom or integer, using name/2). In order to pass a non-atomic Prolog term as an argument to a command, you need to enclose the word in the meta-characters { and }. For example, the flag command can be used to set a global variable. For example:

```
16 |: flag foo bar
```

sets the flag **foo** to the value **bar**. As an other example,

```
17 |: flag foo bar(1,2,3)
```

sets the flag **foo** to the Prolog atom ’bar(1,2,3)’. Finally,

```
17 |: flag foo {bar(1,2,3)}
```

sets the flag **foo** to the complex Prolog term bar(1,2,3), i.e. a term with functor bar/3 and arguments 1, 2 and 3.

Also note that in case the {,} meta-characters are used, then the variables occurring in words take scope over the full command-line. For instance, the parse command normally takes a sequences of words:

```
18 |: parse John kisses Mary
```

In other to apply the parser on a sequence of three variables, where the first and third variable are identical, you give the command:

```
19 |: parse {A} {B} {A}
```

The special meaning of the {,} meta-characters can be switched off by putting a backslash in front of them. For example:

```
53 |: tcl set jan { piet klaas }
=> piet klaas

54 |: tcl puts jan
jan
```

The following meta-devices apply: All occurences of $word are replaced by the definition of the alias **word**. The alias command itself can be used to define aliases:

```
19 |: alias hallo ! cat hallo
20 |: $hallo
```

so command number 20 will have the same effect as typing

```
33 |: ! cat hallo
```

and if this command had been typed as command number 33 then typing

```
35 |: $33
```

gives also the same result.

Moreover, if no alias has been defined, then it will apply the last command that started with the name of the alias:

```
66 |: parse john kisses mary
67 |: $parse
```

Both commands will do the same task.

It is possible to add commands to the command interpreter. The idea is that you can define further clauses for the multifile predicate hdrug_command/3. The first argument is the first word of the command. The second argument will be the resulting Prolog goal, whereas the third argument is a list of the remaining words of the command (the arguments to the command).

```
:- multifile hdrug_command/3.
```

```
hdrug_command(plus,(X is A+B, format('~w~n',[X])),[A,B]).
```

Relevant help information for such a command should be defined using the multifile predicate hdrug_command_help/3. The first argument of this predicate should be the same as the first argument of hdrug_command. The second and third arguments are strings (list of character codes). They indicate respectively usage information, and a short explanation.

```
:- multifile hdrug_command_help/3.
hdrug_command_help(plus,"plus A B","Prints the sum of A and B").
```

For example, consider the case where we want the command **rx** to restart the Tcl/Tk interface. Furthermore, an optional argument of 'on' or 'off' indicates whether the TkConsol feature should be used. This could be defined as follows:

```
hdrug_command(rx,restart_x,L):-
   rxarg(L).

rxarg([on]) :- set_flag(tkconsol,on).
rxarg([off]) :- set_flag(tkconsol,off).
rxarg([]).

hdrug_command_help(rx,"rx [on,off]",
    "(re)starts graphical user interface with/without TkConsol").
```

3.1. flag Flag [Val]

without Val displays value of Flag; with Val sets Flag to Val

3.2. flag Flag [Val]

without Val displays value of Flag; with Val sets Flag to Val

3.3. % Words

ignores Words (comment). Note that there needs to be a space after %.

3.4. fc Files

fcompiles(Files).

3.5. um Files

use_module(Files).

3.6. el Files

ensure_loaded(Files).

3.7. c Files

compile(Files).

3.8. rc Files

reconsult(Files).

3.9. ld Files

load(Files).

3.10. libum Files

for each File, use_module(library(File)).

3.11. librc Files

for each File, reconsult(library(File)).

3.12. libc Files

for each File, compile(library(File)).

3.13. libel Files

for each File, ensure_loaded(library(File)).

3.14. libld Files

for each File, load(library(File)).

3.15. version

displays version information.

3.16. quit|exit|halt|q|stop

quits Hdrug.

3.17. b

break; enters Prolog prompt at next break level.

3.18. d

debug/0.

3.19. nd

nodebug/0.

3.20. p [Goal]

without Goal: quits command interpreter -- falls back to Prolog prompt with Goal: calls Goal. Normally you will need {} around the Goal. For example:

```
p { member(X,[a,b,c]), write(X), nl }
```

3.21. ! Command

Command is executed by the shell. Note that the space between ! and Command is required.

3.22. alias [Name [Val]]

No args: lists all aliases; one arg: displays alias Name; two args: defines an alias Name with meaning Val.

3.23. help [command|flag|pred|hook] [Arg]

displays help on command Arg or flag Arg or predicate Arg or hook Arg; without Arg prints list of available commands, flags, predicates, or hooks.

3.24. ? [command|flag|pred|hook] [Arg]

displays help on command Arg or flag Arg or predicate Arg or hook Arg; without Arg prints list of available commands, flags, predicates, or hooks.

3.25. listhelp [command|flag|pred|hook]

displays listing of all commands, flags, predicates or hooks respectively; without Class displays all help for all classes.

3.26. spy [Module] Pred

set spypoint on Module:Pred; Pred can either be Fun or Fun/Ar.

3.27. cd [Dir]

change working directory to Dir; without argument cd to home directory.

3.28. pwd

print working directory.

3.29. ls

listing of directory contents

3.30. lt [tk/clig/latex] [Type]

prints lexical hierarchy for Type; without Type, prints lexical hierarchy for top

3.31. x

(re)starts graphical user interface

3.32. nox

halts graphical user interface

3.33. tcl Cmd

calls tcl command Cmd; what is returned by the tcl command will be printed on the screen after the => arrow.

```
75 |: tcl expr 3 * [ expr 5 + 4 ]
=> 27
```

Remember that { and } need to be prefixed with backlash since otherwise the Hdrug shell treats them. For instance

63 |: tcl expr 3+4
=> 7

3.34. source File

sources Tcl source File

3.35. s [Format] [Output] Values

displays Objects with specified Format and Output; cf help on s_format, s_output and s_value respectively.

3.36. i/j/s/w/f [Path]/T

Specifies the Format of the s command.

i write/1;

j print/1 (default);

s semantics (third argument of o/3 object terms)

w words (second argument of o/3 object terms)

f Path display as a feature structure; the optional path is a sequence of attributes separated by colons (it selects the value at that path). The prefix of the path can be a sequence of integers seperated by / in order to select a specific node in the tree: this is only possible of the category is a tree datastructure with functor tree/3 where tree labels are specified in the first argument and lists of daughters are specified in the third argument.

T T is a tree-format, display as a tree with that format. Tree-formats are specified with the hook predicates graphic_path, graphic_label and graphic_daughter.

3.37. user/latex/tk/clig/dot

Specifies the Output of the s command.

user as text to standard output (default);

latex LaTeX; ghostview is used to display result;

tk in the canvas of the graphical user interface;

dot used DOT

clig uses Clig

3.38. ObjSpec/DefSpec/ValSpec

Specifies the Objects to be shown form the s command.

ObjSpec will select a number objects (parser/generator results):

s 2 5 8 specifies the objects numbered 2, 5 and 8

s 4 + specifies the objects number 4 and above

s 3 - specifies all objects up to number 3

s 5 to 12 specifies all objects between 5 and 12

DefSpec will select a user_clause definition:

s l Fun/Ar specifies a listing of the Fun/Ar predicate.

ValSpec will specify a goal, and select an argument of that goal:

s [Module:]Fun/Ar [Pos]

The Module prefix is optional (user module is assumed if not specified); the optional Pos argument selects a specific argument to be printed. If no Pos argument is specified then the full goal is printed. For example, if you have the following predicate defined:

```
x23(f(16),g(17),h).
```

then the following commands are possible:

```
|: s x23/3

x23(f(16),g(17),h)

|: s x23/3 1

h

|: s user:x23/3 2

g17
```

3.39. type [t/x/tk/clig/dot] [Type]

displays type t=tree x=latex tree, tk=tk tree, clig=clig tree, dot=dot tree, none=textual information. No Type implies that top is used.

3.40. ps [Keys]

compares parsers on each sentence with key in Keys; without Keys, compares parsers on all availables sentences;

3.41. psint I J

compares parsers on each sentence with key between I and J

3.42. gs [Keys]

compares generators on each lf with key in Keys; without Keys, compares generators on all available lfs;

3.43. gsint I J

compares generators on each lf with key between I and J

3.44. rt [Parser/Generator]

reset tables for parser/generator comparison for parser Parser or generator Generator; without argument reset tables for all parsers and generators

3.45. sentences

lists all sentences

3.46. lfs

lists all logical forms

3.47. pt

print parser comparison overview

3.48. ptt

print parser comparison tables in detail

3.49. pc Sentence

compares parsers on Sentences

3.50. gc LF

compares generators on LF

3.51. gco ObjNo

compares generators on LF of object ObjNo

3.52. * Sentence

parses Sentence

3.53. parse Sentence

parses Sentence

3.54. - Term

if Term is an integer ObjNo, then generate from LF of object ObjNo; otherwise Term is a semantic representation that is generated from

3.55. generate Term

if Term is an integer ObjNo, then generate from LF of object ObjNo; otherwise Term is a semantic representation that is generated from

3.56. lg [File]

with File, compile_grammar_file(File); without File, compile_grammar.

3.57. rcg [File]

with File, reconsult_grammar_file(File); without File, reconsult_grammar.

3.58. tkconsol

(re)starts graphical user interface with TkConsol feature

3.59. av

shows activity status of parsers and generator

3.60. no [gm] List

with gm, List is a list of generators which are set to inactive status; without gm, List is a list of parsers which are set to inactive status

3.61. yes [gm] List

with gm, List is a list of generators which are set to inactive status; without gm, List is a list of parsers which are set to inactive status

3.62. only [gm] List

with gm, List is a list of the only remaining active generators; without gm, List is a list of the only remaining active parsers;

3.63. sts [Parsers]

graphically displays statistics for Parsers; without Parsers displays statistics for all parsers

4. Global Variables

Hdrug manages a number of ‘global’ variables. A flag consists of a ground key (the ‘global variable’) and a value (arbitrary Prolog term). Flags can be set by means of command-line options, command-interpreter commands, the Options menu, and directly by Prolog predicates.

Global variables are passed on to Tcl/Tk. For this purpose there exists a Tcl/Tk array variable, called ‘flag’. If the graphical user interface is running, then this Tcl/Tk variable is automatically updated when the Prolog flag is altered. The predicate tk_flag/1 can be used to explicitly send the value of a flag to Tcl/Tk.

4.1. generator(Generator)

on/off. determines whether Generator is currently active or not, i.e. whether it will take part in generator comparison runs or not.

4.2. parser(Parser)

on/off. Determines whether Parser is currently active or not, i.e. whether it will take part in parser comparison runs or not.

4.3. application_name

Used by the graphical user interface. Determines which application default file is loaded, and the title of the widget.

4.4. batch_command

This flag can be used to run a command **after** 'hdrug' is initialized and **after** the application is initialized by hdrug_initialization/0. The value of the flag is called as a goal and all solutions are found using a failure-driven loop, after which the program terminates.

4.5. clig_tree_active_nodes

Boolean flag which determines whether nodes of clig_trees should be clickable. This is nice, but for large trees slow.

4.6. blt_graph_lines

on/off. Should we connect dots in a blt_graph widget? Default: off.

4.7. debug

0/1/2. Determines the number and detail of continuation messages. Default: 0 (minimum)

4.8. demo

on/off. If demo=on then the system provides somewhat more information. A short representation of the semantic form of a parse will be shown. Note that during test-suite runs this value is off.

4.9. nodeskip

Integer. This flag determines for LaTeX-based tree output the value of nodeskip that is passed on to pstree. If you don't like the tree that is produced then you might try to increase or decrease this value. If the tree is ugly because nodes are too far apart, decrease this value; if the tree is a mess, increase it.

4.10. object_exists_check

on/off. If parse and generation results are saved as objects (flag object_saving) then the system normally checks whether an equivalent object has already been constructed on the basis of the same parse / generation request. This flag determines wether such a check should be made

4.11. object_saving

This value determines whether parse/generation results should be kept as objects for later inspection. If the value is off, no objects are asserted. If it is semi then for each new parse request older objects are removed. If it is on ojbjects are never removed.

4.12. parser

Atom. Determines which parser is currently the parser to use for parse commands.

4.13. add_help_menu

on/off. determines whether on-line help info must be available through the graphical user interface. Should be switched off if you run Hdrug on the display of a different machine and the connection with that machine is slow (like over a phone-line).

4.14. print_table_total

on/off. determines whether during a parse comparison totals should be displayed after each sentence.

4.15. start_results_within_bound

Integer. Determines for the results_within_bound command (hdrug_stats) at which number of millisecond reporting should start. Default: 100.

4.16. end_results_within_bound

Integer. Determines for the results_within_bound command (hdrug_stats) at which number of millisecond reporting should end. Default: 5000.

4.17. incr_results_within_bound

Integer. Determines for the results_within_bound command (hdrug_stats) with which number of millisecond increment percentages should be shown. Default: 100.

4.18. clig_tree_hspace

Integer. Determines the horizontal width between nodes in CLIG trees.

4.19. clig_tree_vspace

Integer. Determines the vertical width between nodes in CLIG trees.

4.20. tcltk

on/off. Determines whether the graphical user interface starts. Default: on

4.21. tkconsol

on/off. Determines whether or not the tkconsol feature (cf. library(tkconsol) should be used. Default: off.

4.22. top_features

Atom. Determines top category used by the parsers. If the value is ‘vp’, then the predicate user:top(vp,Cat) determines the top category (start symbol)

4.23. useful_try_check

on/off. This flag determines during a test-suite run whether a sentence should be parsed even if a shorter sentence has already been timed-out for the current parser. If the value is on, then the sentence is skipped for the current parser.

4.24. user_clause_expansion

on/off. This flag determines whether term_expansion should be used to expand each clause into a user:user_clause/2 predicate. Default: off. Note that you need a multifile declaration for user:user_clause/2 in each file that you load with this flag on. It is often better to load your grammar files with ‘assertall’ and then explicitly construct the user_clause definitions using the ‘compile_user_clause/[0,1]’ predicate.

4.25. cmdint

on/off. This flag determines whether the command-interpreter should be switched on upon startup. Default: off.~n

4.26. update_array_max

Integer. Indicates how many items are passed on in update_array/2 (this predicate is used to inform the gui about the available predicates, types, lexical entries, test sentences, etc.). The default is 1000.

4.27. hdrug_status

This flag is not meant to be set by an application, but is set by Hdrug to communicate the status of the latest parse/generation attempt. The flag has three possible values: success, out_of_memory, time_out. Every parse/generation starts out with te value 'success'. The latter two values are set in the case of a time out exception and a resource error exception.

5. Graphical User Interface

The Hdrug widget (`http://www.let.rug.nl/~vannoord/Hdrug/Manual/ale1.png`) contains the following sub-widgets (from top-to-bottom).

- MenuBar. The MenuBar contains a number of menubuttons. Each of these menu's is described below.
- ObjectBar. The ObjectBar is initially invisible. Once 'objects' are produced by a parser or generator these objects will be placed on this bar.
- Two scrollable canvases. Each canvas is initially left blank but is used for graphical output such as parse-trees, etc. You can scroll the canvas using the scrollbars, but also using the middle mouse button. The relative size of the canvases can be adapted by dragging the border with the left mouse button, or with the file-enlarge left canvas resp. enlarge right canvas commands.
- TkConsol. If the global variable tkconsol is on, then the TkConsol widget treats standard input, standard output and error output.
- ButtonBar. The ButtonBar contains three buttons that can be used to change the value of the top category, the parser and the generator. The current values of these are also listed next to the corresponding buttons. The checkbox 'new canvas' can be switched on in order for the next graphical output to be directed to a seperate window. Finally the rightmost button is a button that is added by manu applications, and which functions as an 'About' button.

5.1. The MenuBar

- The File Menu. The file menu contains buttons of which the meaning is rather straightforward. The first items can be used to (re)compile grammar files. The detailed meaning is dependent on the application.

The ‘compile prolog file’ button lets you choose a Prolog file which will be compiled. The ‘reconsult prolog file’ button behaves similarly but reconsults the file. The ‘source tk/tcl file’ button can be used to source a Tcl/Tk file. The ‘halt’ button halts the application (really!).

The ‘enlarge left canvas’ and ‘enlarge right canvas’ buttons adapt the relative size of the two canvases.

It is also possible to restart the application. This implies that the graphical environment is restarted, but the application files are not reconsulted. This is useful if you are adding/debugging parts of the graphical interface. Use ‘restart x’ to restart. ‘quit x - keep prolog alive’ allows you to stop the interface, but continue the Prolog session.

- The Debug Menu. The debug menu contains a few buttons that are straightforward interfaces to the corresponding Prolog predicates. It contains the options
- nodebug
- debug
- remove spypoints
- spy predicate
- unspy predicate
- statistics
- The Options Menu. The options menu provides an interface to the global variables.
- The Parse Menu. The parse menu is a more interesting menu, although it consists of only a single button. Pressing this button will present a dialog widget asking you for a sentence. You can either type in a new sentence, or select one of the available test sentences. The available sentences are the sentences that were previously parsed during the current session, or that were listed in the test-suite, or that were the result of generation in the current session. The sentence you type or select will be parsed using the current parser, and the current top category. Some statistical output will be presented on standard output. For each parse result a numbered object is created. Each object is visible as a button on the ObjectBar, allowing you to inspect each object. The maximum number of objects that is built is limited by the max_objects flag.

The option ‘compare parsers’ allows you to parse a single test sentence for each of the active parsers.

Use the option ‘parser selection’ to activate or de-active a particular parser.

- The Generate Menu

generate lf. This option is the inverse of the parse option. Now you are prompted for a logical form, which is subsequently input to the current generator. Again, you can either specify a logical form ‘by hand’ or select a pre-existing logical form. Note that there is (yet) no concept of a test-suite for logical forms.

generate object. This option takes the logical form from the object that you select, and generates from this logical form. Note that other information ofthis object is not taken into account in the generation process.

generator selection. Use this option to activate or de-activate a particular generator.

compare generators on lf. Generate from a given logical form for each of the active generators.

compare generators on object. Generate from the logical form of a given object for each of the active generators.

- The Test-suite Menu

The test-suite menu contains a number of options in order to test the application for a (pre-defined) set of sentences. This set of sentences is defined in a file called {suite.pl} in the application directory, and consists of a number of Prolog clauses for the predicate sentence/2, where the first argument is a unique identifier of that sentence, and the second argument is a list of atoms. Note that there is (yet) no concept of a test-suite for logical forms.

The following options are provided.

run test-suite. If this option is chosen then all sentences are parsed in turn, for each of the parsers that are ‘active’}. Use the ‘parser selection’ option to select the parsers you want to include/exclude for the run.

run test-suite and view. This option behaves similar to the previous option, but in addition statistical information comparing the different parsers is shown in a separate TK widget. The statistical information is updated after a sentence has been parsed by all active parsers.

reload test-suite. Choosing this option reloads the test-suite. Note that it does **not** destroy existing test-results.

view test-results. This option contains a number of sub-options that allow you to view the test-results in various ways.

individual tk. Presents a graph in which the length of the sentence is plotted against the parse time, for each of the different parsers and sentences.

totals per #words tk. Similarly, but now averages per sentence length are used.

totals per #words latex. This produces an Xdvi window containing a table of the parse results, again averaged over sentence-length. Note that all :atex files are placed in a temporary directory, given by the environment variable $TMPDIR. If this variable is not set, the directory /tmp is used. The predicate latex:files/5 can be used to get the actual file names that are used.

totals per #readings latex. This produces a table of the parse results, averaged over the number of readings.

individual prolog. This simply gives a Prolog listing of the table_entry/6 predicate.

totals per #words prolog. Prolog output of the average cputimes per parser per #words. This is given as a list of terms t(Length,Time,Parser) with the obvious interpretation.

destroy test results. Removes the test-results, i.e. retracts all clauses of the table_entry/6 predicate.

- The View Menu. The ‘view menu’ contains several sub-menus indicating the type of things you can view. Typically, you can view ‘objects’ (the result of parsing and generation) ‘predicates’ (Prolog predicates) and ‘types’ (if your application uses library(hdrug_feature). For each of the sub-menus you can choose between different output widgets: Tk, CLiG, Prolog, DOT and LaTeX. Finally, for each combination of ‘thing’ and ‘output widget’ you can choose between a number of different output filters. The choice of filter determines whether the output is printed as a tree, a feature structure or the internal Prolog encoding.

Note that not all view options will produce results. Sometimes these options are only defined for particular inputs. For example, in the Tree Adjoining Grammar application there are parsers that build derivation trees, and there are parsers that build derived trees. Hdrug is not always able to tell in advance whether e.g. the Tree(dt) filter (for derivation trees) is defined for a particular object.

- The Help Menu. The help menu contains an interface to the various on-line help resources. The About button produces a rather ugly picture of the author of Hdrug. The Version button produces some information concerning the version of SICStus, Hdrug and application. Finally some applications add an extra ‘About the Grammar’ option that mentions the authors of the application.

5.2. The ObjectBar

- ObjectBar. The results of parsing and generation are asserted in the database as ‘objects’. The existence of objects is indicated in the ObjectBar. For example, if we have parsed the sentence ‘jan kust marie’ in an application, then the ObjectBar contains two buttons, labeled ‘1’ and ‘2’, indicating the first and second parse result. Pressing the button gives rise to a submenu containing three options. The first option allows inspection of the object. The possibilities are essentially those described above under the ‘view menu’. The second and third option allow the generation from the logical form representation of the object (if the application in fact defines a generator), either for the current generator, or for all active generators.

5.3. The ButtonBar

- ButtonBar. The ButtonBar contains maximally three buttons that can be used to change the value of the top category, the parser and the generator. The current values of these are also listed next to the corresponding buttons. Note that there is only a ‘generator’ button if there are generators defined; similarly the ‘parser’ button might not exist in some applications.

6. Interfacing Hdrug

For an application to work with the Hdrug system, there are a number of predicates you have to supply. Furthermore, you can extend the Hdrug system with application-specific options. Finally, you can always overwrite existing Hdrug definitions. In this chapter I discuss the various possibilities.

Parsers and Generators

In Hdrug you can define any number of parsers and generators. A parser and generator is identified by an atomic identifier. A parser is declared by the following directive:

```
:- flag(parser(Identifier),on/off).
```

Similarly, a generator is declared by:

```
:- flag(generator(Identifier),on/off).
```

This defines a parser of generator and moreover tells Hdrug whether this parser is active (on) or not (off). Only if a parser is active, it will be used in parser-comparison runs. Not only should the application define which parsers and generators exist, but usually it will also define the ‘current’ parser and generator. This is achieved by initializing the **parser** and **generator** flag.

```
:- initialize_flag(parser,Identifier).
:- initialize_flag(generator,Identifier).
```

Summarizing, there exist a number of parsers. A subset of those parsers are active. One of the parsers is the current parser.

If a parser (generator) is defined, then there should be a module with the same name which provides the following predicates. Note that only the first one of these predicates, parse/1 or generate/1, is obligatory. The others are not.

- parse/1;generate/1. This predicate is the predicate that does the actual parsing (generation). At the time of calling, the argument of the parse/1 (generate/1) predicate is a term o(Obj,Str,Sem) where Obj is a term in which the top-category is already instantiated. Furthermore, part of the term might be instantiated to some representation of the input string (in case of parsing if the predicate phonology/2 is defined) or some representation of the input logical form (in case of generation if the predicate semantics/2 is defined). But note that the string and logical form are also available (if instantiated) in the second and third argument of the o/3 term.

- count/0.This optional predicate is thought of as a predicate that might produce some statistical information e.g. on the number of chart edges built. Note that library(hdrug_util) contains predicates to count the number of clauses for a given predicate.

- count/1. Similarly, but this time the argument should get bound to some integer. The argument of this predicate determines the final argument of the table_entry/6 predicate in test runs.

- clean/0. If the parser adds items, chart edges etc. to the database, then this predicate defines the way to remove these again.

Top categories

Usually a grammar comes with a notion of a ‘start symbol’ or ‘top category’. In Hdrug there can be any number of different top categories. These top categories are Prolog terms. Each one of them is associated with an atomic identifier for reference purposes. Each top category is defined by a clause for the predicate top/2, where the first argument is the atomic identifier and the second argument is the top-category term. For example:

```
top(s,node(s,_)).
top(np,node(np,_)).
```

The flag ‘top_features’ is used to indicate what the current choice of top-category is. Usually an application defines a default value for this flag by the directive:

```
:- initialize_flag(top_features,Identifier).
```

The identifier relates to the first argument of a top/2 definition.

Strings and Semantics

The predicate semantics/2 defines which part of an object contains the semantics (if any). For example, in an application categories are generally of the form node(Syn,Sem). Therefore, the following definition of semantics/2 is used:

```
semantics(node(_,Sem),Sem).
```

The predicate is mainly used for generation. By default, the predicate is defined as semantics(_,_).

In a similar way, the predicate phonology(Node,Phon) can be defined. This is only useful for 'sign-based' grammars in which the string to be parsed is considered a part of the category. The default definition is phonology(_,_).

The predicate extern_sem/2 can be used to define a mapping between 'internal' and 'external' formats of the semantic representation. This predicate is used in two ways: if a semantic representation is read in, and if a semantic representation is written out. The first argument is the external representation, the second argument the internal one. The default definition is extern_sem(X,X).

Grammar compilation

Currently, the grammar menu contains four distinct options to recompile (parts of) the grammar. It is assumed that if an application is started, the grammars are already compiled. These options will thus be chosen if the grammar has to be recompiled (e.g. because part of the grammar has been changed).

The following four predicates have to be provided by the application. If these predicates do not fulfill your needs, you can always extend the grammar menu (cf. below), or even overwrite it (as in the Ale application).

- compile_grammar/0 should recompile the whole grammar.
- reconsult_grammar/0 should recompile the whole grammar. If files are to be loaded, then 'reconsult' is used rather than 'compile'. This allows easier debugging.
- compile_grammar_file/1 should recompile the grammar file that is its argument.
- reconsult_grammar_file/1 idem, but uses reconsult

Test-suites

A test suite consists of a number of Prolog clauses for the predicate sentence/2, where the first argument is a unique identifier of that sentence, and the second argument is a list of atoms; and clauses for the predicate lf/2. For example:

```
sentence(a,[john,kisses,mary]).
sentence(b,[john,will,kiss,mary]).
lf(1,fut(kiss(john,mary))).
lf(2,past(kiss(mary,john))).
```

The test suite might also contain a definition of the predicate user_max/2. This predicate is used to define an upper time limit, possibly based on the length of the test sentence (the first argument), for parsing that sentence in a test-suite run. By default, Hdrug behaves as if this predicate is defined as follows:

```
user_max(L,Max) :-
    Max is 10000 + (L * 300).
```

Statistical information for each parse is preserved by the dynamic predicate table_entry/6. The arguments of this predicate indicate:

- an atom (the unique identifier of the sentence)
- an integer (the length of the sentence)
- an integer (the number of parses of the sentence, i.e. the degree of ambiguity)
- an atom (the name of the parser)
- an integer (the amount of milliseconds it took to parse the sentence. In case of time-out the atom ‘time_out’).
- a term (often used to indicate the number of chart-edges built). It is determined by the count/1.

Extending the Graphical User Interface

It is easy to extend the Graphical User Interface for a specific application. There are two predicates that you can define. The first predicate, gram_startup_hook_begin/0 is called before loading of hdrug.tcl, whereas the predicate gram_startup_hook_end/0 is called at the end of the loading of this file.

Viewing Prolog Clauses

If you want to use Hdrug’s built-in facilities to view Prolog clauses, then it is neccessary that these clauses are accessible via the predicate user_clause/2. The arguments of this predicate are the head and the body of the clause respectively. The reason that Hdrug does not use the built-in clause/3 predicate, is that this predicate is only available for dynamic clauses.

The easiest way to obtain user_clause/2 definitions is to turn on a term_expansion definition with the appropriate effect. This is done by setting the user_clause_expansion flag to on.

6.1. use_canvas(+Mode,LeftRightTop)

Mode is a term indicating the type of data-structure to be displayed. It is one of tree(TreeMode), fs, text, chart, stat. The predicate should instantiate the second argument as one of the atoms left, right or top (for a new widget).

6.2. help_hook(PredSymbol,UsageString,ExplanationString)

This predicate can be defined to provide help on a hook predicate with predicate symbol PredSymbol. The UsageString is a list of character codes which shortly shows the usage of the predicate. The help_hook predicate which is defined for the help_hook predicate itself has as its UsageString "help_hook(PredSymbol, UsageString, ExplanationString)". The ExplanationString is a list of charactercodes containing further explanation.

6.3. ParserModule:parse(o(Cat,Str,Sem))

If ParserModule is the current parser, then this predicate is called to do the actual parsing. At the time of calling, the argument of the parse/1 predicate is a term o(Obj,Str,Sem) where Cat is a term in which the top-category is already instantiated. Furthermore, part of the term may have been instantiated to some representation of the input string (if the hook predicate phonology/2 was defined to do so). The input string is also available in the second argument of the o/3 term. The third argument is not used for parsing.

6.4. GeneratorModule:generate(o(Cat,Str,Sem))

If GeneratorModule is the current generator, then this predicate is called to do the actual generation. At the time of calling, the argument of the generate/1 predicate is a term o(Obj,Str,Sem) where Cat is a term in which the top-category is already instantiated. Furthermore, part of the term may have been instantiated to some representation of the input semantics (if the hook predicate {semantics/2} was defined to do so). The input semantics is also available in the third argument of the o/3 term. The second argument is not used for generation.

6.5. Module:count

This optional predicate is thought of as a predicate that might display some statistical information e.g. on the number of chart edges built. The predicate Module:count is called in module Parser after parsing has been completed for parser Parser or it is called in module Generator after generation has been completed for generator Generator. Note that library(hdrug_util) contains predicates to count the number of clauses for a given predicate.

6.6. Module:count

This optional predicate is thought of as a predicate that might display some statistical information e.g. on the number of chart edges built. The predicate Module:count is called in module Parser after parsing has been completed for parser Parser or it is called in module Generator after generation has been completed for generator Generator. Note that library(hdrug_util) contains predicates to count the number of clauses for a given predicate.

6.7. Module:clean

This optional predicate is thought of as a predicate that might remove e.g. chart edges added dynamically to the database once parsing has been completed. The predicate Module:clean is called in module Parser after parsing has been completed for parser Parser or it is called in module Generator after generation has been completed for generator Generator.

6.8. start_hook(parse/generate,Module,o(A,B,C),Term)

This predicate is a hook that is called before the parser starts. Its first argument is either the atom parse or the atom generate; the second argument is the current parser or generator (hence the name of the module); the third argument is an object. The fourth argument can be anything. It wis provided to pass on arbitrary information to the result_hook and end_hook hook predicates. For example, the predicate could pass on information concerning the current memory usage of Sicstus. This information could then be used by end_hook to compute the amount of memory that the parser has consumed. The time required by the start_hook predicate is NOT considered to be part of parsing time; cf start_hook0/4 for a similar hook predicate of which timing IS considered part of parsing time

6.9. start_hook0(parse/generate,Module,o(A,B,C),Term)

This predicate is a hook that is called before the parser starts. Its first argument is either the atom parse or the atom generate; the second argument is the current parser or generator (hence the name of the module); the third argument is an object. The fourth argument can be anything. It is provided to pass on arbitrary information to the result_hook and end_hook hook predicates. For example, the predicate could pass on information concerning the current memory usage of Sicstus. This information could then be used by end_hook to compute the amount of memory that the parser has consumed. The time required by the start_hook0 predicate IS considered to be part of parsing time; cf start_hook/4 for a similar hook predicate of which timing is NOT considered part of parsing time

6.10. result_hook(parse/generate,Module,o(A,B,C),Term)

This predicate is a hook that is called for each time the parser or generator succeeds. Its first argument is either the atom parse or the atom generate; the second argument is the current parser or generator (hence the name of the module); the third argument is an object. The fourth argument can be anything. It is provided to pass on arbitrary information from the

start_hook hook predicate. Warning: the time taken by result_hook will always be considered as part of the time required for parsing. Consider using the demo flag to ensure that expensive result_hooks are not fired for parsing comparison runs.

6.11. end_hook(parse/generate,Module,o(A,B,C),Term)

This predicate is a hook that is called if the parser / generator can not wfind any results anymore. Its first argument is either the atom parse or the atom generate; the second argument is the current parser or generator (hence the name of the module); the third argument is an object. The fourth argument can be anything. It is provided to pass on arbitrary information from the start_hook hook predicate. Note that at the moment of calling this predicate the object will typically NOT be instantiated. The time required by end_hook is NOT considered to be part of parsing time; see end_hook0.

6.12. end_hook0(parse/generate,Module,o(A,B,C),Term)

This predicate is a hook that is called if the parser / generator can not find any results anymore. Its first argument is either the atom parse or the atom generate; the second argument is the current parser or generator (hence the name of the module); the third argument is an object. The fourth argument can be anything. It is provided to pass on arbitrary information from the start_hook hook predicate. Note that at the moment of calling this predicate the object will typically NOT be instantiated. The time required by end_hook0 IS considered to be part of parsing time; see end_hook0.

6.13. top(Name,Cat)

Usually a grammar comes with a notion of a ‘start symbol’ or ‘top category’. In Hdrug there can be any number of different top categories, of which one is the currently used top category. These top categories are Prolog terms. Each one of them is associated with an atomic identifier for reference purposes. Each top category is defined by a clause for the predicate top/2, where the first argument is the atomic identifier and the second argument is the top-category term. The latter term will be unified with the first argument of the o/3 terms passed on to parsers and generators.

```
top(s,node(s,_)).
top(np,node(np,_)).
```

The flag ‘top_features’ is used to indicate what the current choice of top-category is. Usually an application defines a default value for this flag. The identifier relates to the first argument of a top/2 definition.

6.14. semantics(Cat,Sem)

The predicate semantics/2 defines which part of an object contains the semantics (if any). For example, if in an application categories are generally of the form node(Syn,Sem), then the following definition of semantics/2 is used:

```
semantics(node(_,Sem),Sem).
```

The predicate is mainly used for generation.

6.15. phonology(Cat,Phon)

This predicate is useful for ‘sign-based’ grammars in which the string to be parsed is considered a part of the category. This predicate is called before parsing so that in such cases the current string Phon can be unified with some part of the object.

6.16. extern_sem(Extern,Intern)

This predicate can be defined in order to distinguish internal and external semantic representations. This predicate is used in two ways: if a semantic representation is read in, and if a semantic representation is written out. The first argument is the external representation, the second argument the internal one. The default definition is extern_sem(X,X). A typical usage of this predicate could be a situation in which an external format such as kisses(john,mary) is to be translated into a feature structure format such as [pred=kisses, arg1=john, arg2=mary]. NB, the external format is read in as a single Prolog term.

6.17. extern_phon(Extern,Intern)

This predicate can be defined in order to distinguish internal and external phonological representations. This predicate is used in two ways: if a phonological representation is read in, and if a phonological representation is written out. The first argument is the external representation, the second argument the internal one. The default definition is extern_phon(X,X). NB, the external format is read in as a list of Prolog terms.

6.18. sentence(Key,Sentence), sentence(Key,Max,Sentence)

Applications can define a number of test sentences by defining clauses for this predicate. For ease of reference, Key is some atomic identifier (typically an integer). Sentence is typically a list of atoms. The parser comparison predicates refer to this atomic identifier. Example sentences are also listed in the listbox available through the parse menu-button. Max can be an integer indicating the maximum amount of milliseconds allowed for this sentence in parser comparison runs.

6.19. lf(Key,LF), lf(Key,Max,Lf)

Applications can define a number of test logical forms by defining clauses for this predicate. For ease of reference, Key is some atomic identifier (typically an integer). LF is a term (external format of a logical form). The generator comparison predicates refer to this atomic identifier. Example logical forms are also listed in the listbox available through the generate menu-button. Max can be an integer indicating the maximum amount of milliseconds allowed for this lf in generator comparison runs.

6.20. user_max(Length,Max)

This predicate is used to define an upper time limit, possibly based on the length of the test sentence (the first argument), for parsing that sentence in a test-suite run. By default, Hdrug behaves as if this predicate is defined as follows: user_max(L,Max) :- Max is 10000 + (L **L** 300). If you don't want a time out at all, then define this predicate as user_max(_,0).

6.21. gram_startup_hook_begin

This predicate is meant to be used to extend the graphical user interface. It is called right before Hdrug's own graphical user interface definitions are loaded (i.e., right before hdrug.tcl is sourced).

6.22. gram_startup_hook_end

This predicate is meant to be used to extend the graphical user interface. It is called right after Hdrug's own graphical user interface definitions are loaded (i.e., right after hdrug.tcl is sourced). A typical use is to add application specific menu-buttons, etc.

6.23. user_clause(Head,Body)

If you want to use Hdrug's built-in facilities to view Prolog clauses, then it is neccessary that these clauses are accessible via the predicate user_clause/2. The arguments of this predicate are the head and the body of the clause respectively. Note that the body of the clause should be provided as a list of goals, rather than a conjunction. The reason that Hdrug does not use the built-in clause/3 predicate, is that this predicate is only available for dynamic clauses. The easiest way to obtain user_clause/2 definitions is to turn on a term_expansion definition with the appropriate effect; cf flag(user_clause_expansion).

6.24. graphic_path(Format,Obj,Term)

One of the three hook predicates which together define tree formats. The others are graphic_label/3 and graphic_daughter/4. The Hdrug libraries contain extensive possibilities to produce output in the form of trees. Only a few declarations are needed to define what things you want to see in the tree. In effect, such declarations define a 'tree format'. In Hdrug, there can be any number of tree formats. These tree formats are named by a ground identifier. A

tree format consists of three parts: the path definition indicates what part of the object you want to view as a tree; the label definition indicates how you want to print the node of a tree; and the daughter definition indicates what you consider the daughters of a node. The graphic_path definition is the first part. For instance if the parser creates an object of the form node(Syn,Sem,DerivTree) where DerivTree is a derivation tree, then we can define a tree format ‘dt’ where the graphic_path definition extracts the third argument of this term: graphic_path(dt,node(_,_,Tree),Tree).

6.25. graphic_label(Format,Node,Label)

One of the three hook predicates which together define tree formats. The others are graphic_path/3 and graphic_daughter/4. The Hdrug libraries contain extensive possibilities to produce output in the form of trees. Only a few declarations are needed to define what things you want to see in the tree. In effect, such declarations define a ‘tree format’. In Hdrug, there can be any number of tree formats. These tree formats are named by a ground identifier. A tree format consists of three parts: the path definition indicates what part of the object you want to view as a tree; the label definition indicates how you want to print the node of a tree; and the daughter definition indicates what you consider the daughters of a node. The graphic_label definition is the second part. For instance, if subtrees are of the form tree(Node,Ds), where Node are terms representing syntactic objects such as np(Agr,Case) and vp(Agr,Subcat,Sem) then a tree format could be defined which only displays the functor symbol: graphic_label(syn,tree(Term,_),Label) :- functor(Term,Label,_).

6.26. graphic_daughter(Format,No,Term,Daughter)

One of the three hook predicates which together define tree formats. The others are graphic_label/3 and graphic_daughter/4. The Hdrug libraries contain extensive possibilities to produce output in the form of trees. Only a few declarations are needed to define what things you want to see in the tree. In effect, such declarations define a ‘tree format’. In Hdrug, there can be any number of tree formats. These tree formats are named by a ground identifier. A tree format consists of three parts: the path definition indicates what part of the object you want to view as a tree; the label definition indicates how you want to print the node of a tree; and the daughter definition indicates what you consider the daughters of a node. The graphic_daughter definition is the third part. For instance if subtrees are of the form tree(Label,Daughters), where Daughters is a list of daughters, then you could simply define: graphic_daughter(syn,No,tree(_,Ds),D):- lists:nth(No,Ds,D).

6.27. show_node(Format,Node)

If trees are displayed on the canvas widget, then it is possible to define an action for clicking the left-most mouse button on the node of the tree. This action is defined by this predicate. Format is the identifier of a tree format, and Node is the full sub-tree (that was used as input to the graphic_label definition).

6.28. show_node2(Format,Node)

If trees are displayed on the canvas widget, then it is possible to define an action for clicking the middle mouse button on the node of the tree. This action is defined by this predicate. Format is the identifier of a tree format, and Node is the full sub-tree (that was used as input to the graphic_label definition).

6.29. show_node3(Format,Node)

If trees are displayed on the canvas widget, then it is possible to define an action for clicking the rightmost mouse button on the node of the tree. This action is defined by this predicate. Format is the identifier of a tree format, and Node is the full sub-tree (that was used as input to the graphic_label definition).

6.30. tk_tree_user_node(Label,Frame)

If a tree-format is defined which matches user(_), then if a tree is to be displayed on the Canvas widget this predicate is responsible for creating the actual nodes of the tree. Label is the current label, and Frame is the identifier of a Tcl/Tk frame which should be further used for this label. The frame is already packed.

6.31. clig_tree_user_node(Label)

If a tree-format is defined which matches user(_), then if a tree is to be displayed using Clig output, then this predicate is responsible for creating the actual nodes of the tree. Label is the current label.

6.32. dot_tree_user_node(Label)

If a tree-format is defined which matches user(_), then if a tree is to be displayed using DOT output, then this predicate is responsible for creating the actual label of the nodes of the tree. Label is the current label.

6.33. latex_tree_user_node(Label)

If a tree-format is defined which matches user(_), then if a tree is to be displayed using LaTeX output, then this predicate is responsible for creating the actual nodes of the tree. Label is the current label.

6.34. shorten_label(Label0,Label)

This predicate can be defined for feature-structure display of tree nodes; its intended use is to reduce the information of a given node.

6.35. call_build_lab(F,Fs,L)

for library(hdrug_call_tree)

6.36. call_build_lab(Functor/Arity)

for library(hdrug_call_tree)

6.37. exceptional_sentence_length(Phon,Length)

For (internal) phonological representations this predicate can be defined to return the length of the representation. If the predicate is not defined, then the representation is assumed to be a list, and the length is assumed to be the number of elements of the list. The length of phonological representations is used by the display of the results of parser comparison runs.

6.38. exceptional_lf_length(Sem,Length)

For (internal) semantic representations this predicate can be defined to return the length of the representation. If the predicate is not defined, then the representation is assumed to be a term, and the length is assumed to be the number of characters required to print the term. The length of semantic representations is used by the display of the results of generator comparison runs.

6.39. hdrug_initialization

If hdrug is started, then three things happen. First, hdrug treats its command line options. After that, the predicate hdrug_initialization is called. Finally, the graphical user interface is started (if flag(tcltk) is on). This predicate can thus be used to define application-specific initialization.

6.40. hdrug_command(Name,Goal,Args)

This predicate can be used to define further commands for the command interpreter. Name is the first word of the command, Goal is the resulting Prolog goal, and Args is a possibly empty list of arguments to the command.

6.41. hdrug_command_help(Name,UsageString,ExplanationString)

This predicate can be used to provide help information on commands for the command interpreter. Name is the first word of the command, The second argument displays usage information in a short form (list of character codes); the third argument is a list of character codes containing an explanation of the command.

6.42. help_flag(Flag,Help)

This predicate can be used to provide help information on global variable Flag. Help is a list of character codes containing the help info.

6.43. option(Option,ArgvIn,ArgvOut)

This predicate can be used to define application-specific command-line options to the hdrug command. Option is the option minus the minus sign; moreover Option relates to the first argument of a corresponding usage_option/3 definition. The second and third argument is a difference list of the list of options in case the option takes further arguments.

6.44. usage_option(Option,UsageString,ExplanationString)

This predicate is defined to provide help information on the Option startup option (cf. option/3). The UsageString is a list of character codes presenting short usage information; ExplanationString is a list of character codes containing the explanation of the option.

6.45. tk_tree_show_node_help(TreeFormat,Atom)

If a tree according to TreeFormat is displayed on the canvas, then this predicate can be defined in order that below the widget a short message appears indicating what actions are bound to clicking on the tree nodes. Atom is the message.

6.46. show_relation(F/A)

you can define the relation show_relation/1 to define an action for pressing the first mouse-button on a relation name, when viewing predicate definitions in the Tk Canvas. The argument is a Functor/Arity pair. For example,

```
show_relation(F/A) :-
        show_predicate(F/A,fs,tk).
```

will show the predicate definition.

6.47. display_extern_sem(+ExtSem)

Predicate to print a given external format of semantics.

6.48. display_extern_phon(+ExtPhon)

Predicate to print a given external format of phonology.

6.49. compile_test_suite(+File)

Predicate to compile the test suite in file File.

6.50. reconsult_test_suite(+File)

Predicate to reconsult the test suite in file File.

6.51. show_object_default2(+Int)

Predicate which is called if the user presses mouse button <2> on the object button number Int. A typical definition could be, for instance:

```
show_object_default2(No):-
   show_object_no(No,tree(syn),clig).
```

6.52. show_object_default3(+Int)

Predicate which is called if the user presses mouse button <2> on the object button number Int.

7. Command-line Options

When Hdrug is started, it first interprets the command-line options. Command-line options are interpreted from left to right. The following section lists the command-line options which are standard. Each application possibly extends this list: this is defined below.

Note that command-line options are interpreted **before** application-specific initialization is performed. This is to allow command-line options to have an effect on this initialization. Refer to the hook predicate **hdrug_initialization** for application-specific initialization.

Hdrug applications can extend the list of possible startup options by adding definitions to the multifile predicate option/3. Short usage information for such options can be defined with further definitions for the multifile predicate usage_option/3. An an example, the following definitions ensure that an option -rc File will reconsult the file File:

```
:- multifile option/3, usage_option/3.

option(rc) --> [File], { reconsult(File) }.
usage_option(rc,"rc File","File is reconsulted.").
```

7.1. -flag Att Val

Sets global variable Att to Val; Val is read as an atom. Consider using the Flag=Val option if you want to assign arbitrary Prolog terms to Att.

7.2. -iflag Att Val

Sets global variable Att to Val; Val is read as an integer. Equivalent to Flag=Val where Val is an integer.

7.3. -pflag Att Val

Sets prolog_flag(Att) to Val; Val is read as an atom. This is an interface to the SICStus Prolog built-in predicate prolog_flag/3.

7.4. -flag Att Val

Sets global variable Att to Val; Val is read as an atom. Consider using the Flag=Val option if you want to assign arbitrary Prolog terms to Att.

7.5. -cmd Goal

evaluates Prolog Goal; Goal is parsed as Prolog term. Example:

```
hdrug -notk -cmd 'listing(library_directory)' -quit
```

7.6. -tk

Indicates that the graphical user interface should be started when hdrug starts. Equivalent to tcltk=on. This is the default.

7.7. -notk

Indicates that the graphical user interface should not be started when hdrug starts. Equivalent to tcltk=off. The default is to start the graphical user interface.

7.8. -dir Dir

This options ensures that Dir is added to the list of library directories.

7.9. -help

This display usage information and terminates.

7.10. -l File

Loads the file File (containing Prolog), using the goal use_module(File).

7.11. -parser Parser on/off

This option indicates that the parser Parser is set to on (off). Parsers which are on will take part in parser comparison runs.

7.12. -generator Generator on/off

This option indicates that the generator Generator is set to on (off). Generators which are on will take part in generator comparison runs.

7.13. -quit

Terminates Hdrug. Useful in combination with the -cmd Goal option.

8. List of Predicates

This chapter lists the important predicates used in Hdrug.

8.1. concat(Atom,Atom,Atom)

Two of the three arguments must be Prolog atoms. The print-name of the third atom is the concatenation of the print names of the first two atoms. Examples:

```
| ?- concat(foo,bar,X).

X = foobar ?

yes
| ?- concat(X,bar,foobar).

X = foo ?

yes
| ?- concat(foo,X,foobar).
```

X = bar ?

8.2. concat_all(+ListOfAtoms,?Atom[,+Atom])

concetenates the print names of all the atoms in ListOfAtoms together; possibly using the optional third argument as a seperator. Example:

?- concat_all([foo,bar,foo,bar],L,'+').

L = 'foo+bar+foo+bar' ?

8.3. between(+Lower, +Upper, ?Number[, +/-])

Is true when Lower, Upper, and Number are integers, and Lower =< Number =< Upper. If Lower and Upper are given, Number can be tested or enumerated. If either Lower or Upper is absent, there is not enough information to find it, hence failure. Numbers are generated in ascending order. If you want descending order, use between/4. The optional fourth argument is the atom + to indicate ascending order, or

- to indicate descending order. Example:

?- findall(X,between(1,10,X), Xs).

Xs = [1,2,3,4,5,6,7,8,9,10] ?

?- findall(X,between(1,10,X,-), Xs).

Xs = [10,9,8,7,6,5,4,3,2,1] ?

8.4. atom_term(+Atom,?Term).

Atom is read-in as if it where a Prolog term. Example:

| ?- atom_term('f(A,B,A)',L).

L = f(_A,_B,_A) ?

8.5. term_atom(+Term,?Atom).

The Prolog term Term is turned into an atom, as if quotes were placed around it. Example:

| ?- term_atom(f(f(f(f))),L).

L = 'f(f(f(f)))' ?

As is clear from the following example, the result is arbitrary in case Term contains variables:

```
?- term_atom(f(_A,_B,_A),L).

L = 'f(_83,_105,_83)' ?
```

8.6. gen_sym(-Atom[,+Prefix])

A new atom Atom is generated. If Prefix is specified, then the print name of Atom will start with Prefix.

8.7. report_count_edges_pred(:Spec)

Writes to standard output the number of times :Spec succeeds. Example:

```
| ?- report_count_edges_pred(library_directory/1).

library_directory/1: 2
```

8.8. report_count_edges(:Goal)

Writes to standard output the number of times :Goal succeeds. Example:

```
| ?- report_count_edges(lists:member(_,[a,b,c,d])).

lists:member(_95,[a,b,c,d]) : 4
```

8.9. count_edges(:Goal,?Int)

Int is an integer indicating the number of times Goal succeeds.

8.10. debug_call(+Int,:Goal)

If Int is smaller or equal to the current value of flag(debug), then Goal is called. Used to wrap around debugging and continuation calls. Larger values for Int indicate that the goal is executed less often.

8.11. debug_message(+Int,+FormatStr,+FormatArgs)

If Int is smaller or equal to the current value of flag(debug), then the goal format(user_error,FormatStr,FormatArgs) is executed.

8.12. initialize_flag(+Flag,?Val)

Hdrug manages a number of global variables, called flags. This predicate sets flag Flag to Val only if Flag is currently undefined.

8.13. set_flag(+Flag,?Val)

Hdrug manages a number of global variables, called flags. This predicate sets flag Flag to Val.

8.14. flag(+Flag[,?OldVal[,?NewVal]])

Hdrug manages a number of global variables, called flags. This predicate sets flag Flag to NewVal, unifying the old value with OldVal. If only two arguments are given, then the flag is unchanged. If only a single argument is given, then Flag is allowed to be uninstantiated. It will be bound to all existing flags upon backtracking.

8.15. un_prettyvars(+Term0,?Term)

Reverses the effect of prettyvars; i.e. all '$VAR'/1 terms are replaced by corresponding variables.

8.16. prettyvars(?Term)

Similar to the built-in numbervars, except that all variables which only occur once in Term are replaced by '$VAR'('_').

8.17. prolog_conjunction(Conjunction, ListOfConjuncts)

handles the syntax of conjuncts. This code wraps call(_) around variables, flattens conjunctions to (A;(B;(C;(D;E)))) form, and drops 'true' conjuncts.

8.18. prolog_disjunction(Disjunction,ListOfDisjuncts)

handles the syntax of disjuncts. This code wraps call(_) around variables, flattens disjunctions to (A,(B,(C,(D,E)))) form, and drops 'false' disjuncts.

8.19. try_hook(:Goal[,:Goal])

Tries to call Goal, but only if the predicate is known to exist. If the first Goal fails, or if it does not exist, then the second goal is called. If no second goal is given then the predicate succeeds.

8.20. hook(:Goal).

hook/1 calls its argument, but only if it is defined; if it is not defined the precate fails. Useful to call optional hook predicates for which no undefined predicate warnings should be produced.

8.21. if_gui(:Goal[,:AltGoal])

calls Goal only if graphical user interface is currently running; if not the predicate calls AltGoal, if it is specified, or succeeds

8.22. r

Starts the command interpreter.

8.23. start_x

Attempts to start the graphical user interface, but will not start it if flag(tcltk) is switched off

8.24. update_array(+List,+ArrayName)

a Tcl array named ArrayName is constructed where the values in List are to be the values in the array, i.e. ArrayName(1), ArrayName(2), etc.; the special value ArrayName(max) is set to the last index of the array (counting starts at 0). The flag update_array_max can be used to pass to Tcl only the first N items. If that value is 0 then all items are passed on (default=1000).

8.25. tk_fs(+Term)

Term is displayed as a feature-structure on the canvas widget of the graphical user interface

8.26. tk_fs(List)

Each Term in List is displayed as a feature-structure on the canvas widget of the graphical user interface

8.27. tk_term(?Term)

Term is displayed on the canvas of the graphical user interface

8.28. tcl_eval(+Cmd[,-Return])

Abbrevation for the tcltk library predicate tcl_eval/3. The current TclTk interpreter, accessible through the tcl_interp flag, is added as the first argument.

8.29. tcl(+Expr[,+Subs[,-ReturnAtom]])

Expr is a string as accepted as the second argument of format/3; the optional Subs is equivalent to the third argument of format/3. After evaluating the meta-charcters in Expr, the string is sent as a tcl command using the current tcl interpreter (flag tcl_interp). The return string is turned into an atom and available in the optional third argument.

8.30. show_object_no(+No,+Style,+Output)

Displays the object numbered No using the Style and Output. These latter two arguments are of the type accepted by the first and second argument of the generic show/3 predicate.

8.31. show(+Style,+Medium,+Things)

Generic interface to the Hdrug visualization tools. Style is one of:

words (only defined for object/2 things; displays the phonological representation of an object, i.e. Phon in object(Ident,o(Cat,Phon,Sem))

sem (only defined for object/2 things; displays the semantic representation of an object, i.e. Sem in object(Ident,o(Cat,Phon,Sem))

fs(+Path) (extracts the feature structure at path Path; and displays the result as a feature structure in matrix notation. In such a Path the prefix might consist of integers to refer to daughters in a tree/3 tree structure; 0 is the root node of a local tree.)

fs (feature structure in matrix notation)

term(print) output as a Prolog term, using print where appropriate (in order that any application-specific portray/1 hook predicates will be applicable)

term(write) same as term(print), but not using print.

tree(Format) displays as a tree using Format as the relevant tree-format. Such a tree-format is defined by clauses for the hook predicates graphic_path, graphic_daughter and graphic_label.

Medium is one of:

user (normal text to SICStus Prolog standard output).

tk (on a canvas of the graphical user interface).

latex (latex code is input to latex and either xdvi or dvips followed by ghostview).

clig (using the CLiG system).

dot

and Things is a list where each element is one of:

object(Ident,o(Cat,Words,Sem))

value(Term)

clause(Head,Body), Body a list of goals.

8.32. hdrug_latex:latex_tree(+TreeFormat,+Term)

Displays Term as a tree according to the TreeFormat specifications, in Ghostview. This predicate produces LaTeX code (with PsTricks extensions); it runs LaTeX and dvips on the result. The TreeFormat should be specified by means of clauses for the hook predicates graphic_path, graphic_daughter and graphic_label.

8.33. hdrug_latex:latex_tree(+TreeFormat,+ListOfTerms)

Displays each Term in ListOfTerms as a tree according to the TreeFormat specifications, in Ghostview. This predicate produces LaTeX code (with PsTricks extensions); it runs LaTeX and dvips on the result. The TreeFormat should be specified by means of clauses for the hook predicates graphic_path, graphic_daughter and graphic_label.

8.34. hdrug_latex:latex_fs(+Term)

Displays Term as a feature structure in Xdvi. The predicate produces LaTeX code (using Chris Manning's avm macro's); it runs LaTeX and xdvi on the result.

8.35. hdrug_latex:latex_fs_list(+List)

Displays each Term in List as a feature structure in Xdvi. The predicate produces LaTeX code (using Chris Manning's avm macro's); it runs LaTeX and xdvi on the result.

8.36. hdrug_latex:latex_term(+Term)

Displays Term in Xdvi. The predicate produces LaTeX code; it runs LaTeX and xdvi on the result.

8.37. hdrug_latex:latex_term_list(+List)

Displays each Term in List in Xdvi. The predicate produces LaTeX code; it runs LaTeX and xdvi on the result.

8.38. generate(Sem)

generates from the semantic representation Sem. Sem is first filtered through the hook predicate extern_sem.

8.39. parse(Phon)

parses from the phonological representation Phon; typically Phon is a list of atoms, refer to the extern_phon hook predicate for more complex possibilities.

8.40. generate_obj_no(Integer)

generated from the semantic representation of object Integer. Only the semantic representation of that object is passed to the generator.

8.41. available

Lists all available parsers and generators, and their associated activity status. During parser comparison and generator comparison, only those parsers and generators are compared which are currenly active.

8.42. object(No,Object)

Results of parsing and generation are normally added to the database. This predicate can be used to fetch such an object. The first argument is an integer used as the key of the object, the second argument is a triple o(Cat,Phon,Sem).

8.43. reset_table / reset_table(ParGen)

Without an argument, removes all results of parser comparison and generator comparison runs. With an argument, only remove information concerning that particular parser or generator.

8.44. parser_comparisons / parser_comparisons(Keys)

Without arguments, compares active parsers on all sentences in test suite. With an argument, Keys is a list of keys which relate to the first argument of the sentence hook predicate. The active parsers will be compared on sentences with a matching key.

8.45. generator_comparisons / generator_comparisons(Keys)

Without arguments, compares active generators for all logical forms of test suite. With an argument, Keys is a list of keys which relate to the first argument of the lf hook predicate. Only the logical forms with a matching key are compared.

8.46. sentences

lists all sentences in test-suite

8.47. lfs

lists all logical forms in test-suite

8.48. parse_compare(Sentence)/parse_compare(Max,Sentence)

Compares active parsers on Sentence. In the binary format, Max is an integer indicating the maximum amount of msec.

8.49. generate_compare(Lf)/generate_compare(Max,Lf)

Compares active generators on Lf. In the binary format, Max is an integer indicating the maximum amount of msec.

8.50. compile_user_clause[(Module)]

This predicate will construct Module:user_clause/2 definitions based on the available Module:clause/2 clauses (if no Module is specified, user is assumed). In the body of these clauses feature constraints are expanded out. The user_clause predicate is used for graphical display of predicates defined in the grammar. So you have to add a (typically multifile) predicate user:user_clause(A,B) :- Module:user_clause(A,B) for each Module.

9. hdrug_call_tree: Displaying Lexical Hierarchies

This library is intended to be used to display lexical hierarchies in tree format. The relevant predicates all take a unary predicate Pred. The predicates then pretty print in a tree format the hierarchy related to the predicate Fuctor/1 as follows. Pred dominates all predicates that call Pred in their body.

If the optional Functor argument is absent, then the user:call_default/1 hook predicate is used to obtain Functor.

```
transitive(X) :- verb(X).

verb(X) :- lex(X).

noun(X) :- lex(X).
```

gives the tree: lex(verb(transitive),noun)

Other calls in the body are attached to the label, as a poor man's way to illustrate multiple inheritance:

```
transitive(X) :- verb(X).

verb(X) :- lex(X).

noun(X) :- lex(X),other(X).
```

gives: lex(verb(transitive),noun[other])

Leaves of the tree can be defined by the user (e.g. to stop the tree at interesting point, and to give interesting info in the label, use the hook predicate user:call_leaf(Call,Label). And yes, don't forget the obvious: it is assumed that the predicates are not recursive.

9.1. Hook Predicates

This section lists the hook predicates for the hdrug_call_tree library.

9.1.1. user:call_default(Functor)

Indicates that Functor is the default predicate for the various call_tree predicates.

9.1.2. user:call_clause(Head,Body)

9.1.3. user:call_leaf(Leaf)

9.1.4. user:call_build_lab(F,Fs,L)

9.1.5. user:call_ignore_clause(F/A)

9.2. Predicates

This section lists the predicates exported by the hdrug_call_tree library.

9.2.1. hdrug_call_tree:call_tree_bu[_tk/_clig/_latex][(Functor)]

pretty prints in a tree format the hierarchy related to the predicate Fuctor/1. If the optional Functor argument is absent, then the user:call_default/1 hook predicate is used to obtain Functor. The _tk _clig and _latex suffices indicate that a different output medium should be chosen (instead of the console).

10. hdrug_chart: Displaying Charts

The module hdrug_chart is intended to be used to display chart-like data-structures (on a Tcl/Tk canvas).

10.1. Global Variables

This section lists the global variables maintained by the hdrug_chart library.

10.1.1. user:chart_xdist

This flag determines the horizontal distance between nodes of the chart.

10.1.2. user:chart_ydist

This flag determines the distance between edges over the same range.

10.2. Hook Predicates

This section lists the hook predicates for the hdrug_chart library.

10.2.1. user:pp_chart_show_node_help(Atom)

Short atom to be displayed in the help-line upon entering nodes of the chart. This is typically used to indicate the corresponding actions of mouse clicks on the nodes.

10.2.2. user:pp_chart_item[23](Ident)

Used by hdrug_chart. This predicate can be used to define an action to be executed upon clicking the label of an edge of the chart. This variant is for edges above the horizontal axis. The argument Ident refers to the fourth argument of the relevant edge that was one of the elements in the list passed on as the second argument of the pp_chart/3 predicate. The variants with a 2 or 3 suffix are used to define an action for the second or third mouse button.

10.2.3. user:pp_chart_item_b[23](Ident)

Used by hdrug_chart. This predicate can be used to define an action to be executed upon clicking the label of an edge of the chart. This variant is for edges below the horizontal axis. The argument Ident refers to the fourth argument of the relevant edge that was one of the elements in the list passed on as the third argument of the pp_chart/3 predicate. The variants with a 2 or 3 suffix are used to define an action for the second or third mouse button.

10.3. Predicates

This section lists the predicates exported by the hdrug_chart library.

10.3.1. pp_chart(Nodes,Edges,Bedges)

Pretty-printing routine (on the Tk widget) for chart-like datastructures. Nodes is a list of integers, indicating the nodes of the chart (the string positions). Edges is a list of edges. Each edge is a term edge(P,Q,Cat,Ident) where P and Q are chart nodes, Cat is some atom used as the label of the edge, and Ident is some atom used to identify the edge. This identifier is passed on to the hook predicates pp_chart_item and pp_chart_item_b which can be used to define an action for clicking the label of a chart item. Bedges is also a list of edges; these edges are placed on and below the nodes of the chart (this can be used, for instance, to display the words of the chart). Example:

```
?- pp_chart([0,1,2],[edge(0,1,np,1),edge(1,2,vp,2),edge(0,2,s,3)],
                [edge(0,1,jan,4), edge(1,2,slaapt,5)]).
```

11. hdrug_clig: Interface to CLiG

This module provides an interface to Karsten Konrad's CLiG system for visualization of feature-structures and trees.

11.1. Predicates

This section lists the predicates exported by the hdrug_clig library.

11.1.1. clig_fs(Fs)

displays a feature structure in CLiG. Assumes that hdrug(hdrug_feature) is loaded and that feature declarations have been compiled. Example:

```
X:cat => np, clig_fs(value(X)).
```

11.1.2. clig_fs_list(List)

displays each feature structure in List in CLiG. Assumes that hdrug(hdrug_feature) is loaded and that feature declarations have been compiled. Example:

```
X:cat => np, Y:cat => vp, clig_fs_list([X,Y]).
```

11.1.3. clig_tree(Format,Term)

displays a feature structure in CLiG. Format is a tree-format; Term is an arbitrary term. The hook-predicates graphic_path, graphic_label and graphic_daughter are used to obtain the tree structure for Term.

12. hdrug_feature: The Hdrug Feature Library

The feature library provides extensive possibilities to compile feature equations into Prolog terms, and to view such compiled Prolog terms as feature-structures. The motivation for such an approach might be that you want feature structures for readability on the one hand, but Prolog terms and Prolog unification of such terms for effiency reasons internally. The package is heavily influenced by the work of Chris Mellish.

Types

Before feature structures can be compiled into terms, a number of type declarations need to be specified. The declarations that need to be defined are top/1, type/3 and at/1. These three definitions define a type hierarchy. This hierarchy has the shape of a tree. The top/1 definition defines the daughter nodes of the root of the tree. This root is always called ‘top’.

Attributes can be attached to a single type in the type hierarchy. If a type is associated with an attribute then this attribute is inherited by all of its subtypes. The top node of the type hierarchy can be seen as a variable. You can not specify any attributes for this type. The type/3 predicate defines for a given type (first argument) a list of subtypes (second argument) and a list of attributes (third argument).

The at/1 definitions define terminals of the tree that do not introduce attributes. It is an abbreviation of a type/3 definition in which the second and third argument are both the empty list.

As an example, consider the following type tree definition:

```
top([boolean,sign,cat]).
type(boolean,[+,-],[]).
at(+).
at(-).
type(sign,[],[cat,phon,sem]).
type(cat,[noun,verb],[agr]).
type(noun,[],[pro]).
```

```
type(verb,[],[aux,inv,subj]).
```

If this type definition is consulted by Hdrug, and if the directive:

```
:- type_compiler.
```

is called, then it is possible to view the type definition by choosing the ‘view type tk’ menu. This gives rise to a tree on the canvas as
`http://www.let.rug.nl/~vannoord/Hdrug/Manual/type.png`

The meaning of such a type tree can be understood as follows. The class of objects is divided in three mutually exclusive subclasses, called boolean, sign and cat. Objects of type boolean can be further subdivided into classes + or -. Objects of type sign can be further specified for a cat, phon or sem attribute.

The meaning of this type tree can also be understood by looking at the way in which objects of a certain type are represented as Prolog terms. This is illustrated as
`http://www.let.rug.nl/~vannoord/Hdrug/Manual/tree.png`

Equational constraints

If the type definition is compiled, then the following predicates can be used: <=>/2, =>/2, ==>/2. The first predicate equates two **paths**, the second predicate assigns a type to a path, and the third predicate assigns an arbitrary Prolog term to a path.

A path is a Prolog term followed by a sequence of attributes, seperated by a colon (:). Therefore, given the previous example of a type tree, we can have the following equational constraint:

```
X:cat => noun.
   X = sign(_H,cat(noun(_G,_F),_E,_D),_C,_B,_A)
Y:cat:agr <=> Y:cat:subj:cat:agr.
   Y = sign(_O,cat(verb(_N,_M,_L,sign(_K,cat(_J,_E,_I),
                        _H,_G,_F)),_E,_D),_C,_B,_A)
Z:phon ==> [jan,kust,marie].
sign(_D,_C,[jan,kust,marie],_B,_A)
```

Lists

You can add (ordinary Prolog) lists to your type tree by the simple definition:

```
list_type(HeadAtt,TailAtt).
```

This will allow the use of attributes HeadAtt and TailAtt for referring to parts of lists. Furthermore, lists of typed objects will be shown appropriately. For example:

```
[-user].
| list_type(h,t).
| {user consulted, 20 msec 48 bytes}
^D
yes
| ?- X:t:h:cat => verb.

X = [_A,sign(_K,cat(verb(_J,_I,_H,_G),_F,_E),_D,_C,_B)|_L] ?

yes
| ?- X:t:h:cat => verb, show(fs,latex,[value(X)]).
....

X = [_A,sign(_K,cat(verb(_J,_I,_H,_G),_F,_E),_D,_C,_B)|_L] ?
```

Extensionality

Direct subtypes of type ‘top’ are represented using an extra variable position. This is to make sure that objects are only identical if they have been unified. For some types this does not make much sense. Types that you want to consider as ‘extensional’ in this way are to be declared with the predicate extensional/1. Boolean types (cf. below) are extensional by default. Providing an intentional/1 definition makes a boolean type intensional.

The following example illustrates the difference. Without the extensional predicate we have:

```
X:inv => -, X:aux => -, tty_fs(X).
{verb}
|aux <B> {-}
|inv <B>.
```

After declaring that boolean and ‘-’ be extensional types (and recompiling the type tree), we get:

```
X:inv => -, X:aux => -, tty_fs(X).
{verb}
|aux {-}
|inv {-}.
```

The difference is that Hdrug does not show explicitly that the values of aux and inv are the same in the second example. This is redundant information because objects of extensional types always are the same if they have the same information content.

Unify_except

The library provides the predicates unify_except/3, unify_except_l/3 and overwrite/4. The first argument takes two feature terms and a path. The first and second argument are unified **except** for the value at the path}.

As an example (assuming the simple type system given above), we might have:

```
| ?- unify_except(X,Y,cat:agr).

X = sign(_G,cat(_F,_E,_D),_C,_B,_A),
Y = sign(_G,cat(_F,_H,_D),_C,_B,_A) ?
```

The predicate unify_except_l is similar, except it takes a list of paths rather than a single path as its third argument. Finally, the predicate overwrite/4 can be understood by looking at its definition:

```
overwrite(FS,FS2,Path,Type) :-
        unify_except(FS2,FS,Path),
        FS2:Path => Type.
```

Find_type

The meta-logical predicates find_type/2 and find_type/3 can be used to get the most specific type of a feature term. The first argument is the feature term, the second argument is a list of most specific types (for simple usage just consider the first element of this list). The optional third argument is a list of attributes that are appropriate for this type. For example:

```
| ?- X:agr <=> X:subj:agr, find_type(X,[Y|_]).

X = cat(verb(_G,_F,_E,cat(_D,_B,_C)),_B,_A),
Y = verb ?
```

It is clear that find_type/2,3 are meta-logical predicates by looking at the following example, where the conjuncts are swapped:

```
| ?- find_type(X,[Y|_]), X:agr <=> X:subj:agr.

X = cat(verb(_G,_F,_E,cat(_D,_B,_C)),_B,_A),
Y = top ? ;
```

Disjunction and Negation over Atomic Values

A special mechanism is provided for atomic values to allow for disjunction and negation over such atomic values. These atomic values are not declared in the type-system as shown above, but rather they are introduced by the predicate boolean_type/2. The first argument of this predicate is an identifier, the second argument of this predicate is a list of lists that is understood as a set product. For example, agreement features could be defined as:

```
boolean_type(agr,[[1,2,3],[sg,pl],[mas,fem,neut]])
```

So valid and fully specified values for agreement consist of an element from each of the three lists. The syntax for type-assignment is extended to include disjunction (';'), conjunction ('&') and negation ('~') of types. For example, to express that X has either singular masculine or not-second person agreement, we simply state:

```
X => ( sg & mas ; ~2 ).
```

The following example illustrates the use of this package:

```
| ?- [-user].
| boolean_type(agr,[[1,2,3],[sg,pl],[mas,fem,neut]]).
| {user consulted, 10 msec 368 bytes}

yes
| ?- type_compiler.

yes
| ?- X => ( sg & mas ; ~2 ).

X = agr(0,_L,_K,_J,_I,_H,_G,_G,_G,_G,_F,_F,_E,_D,_C,_B,_A,1) ?
```

The example shows how complex terms are created for such boolean types. This is useful because disjunction and negation can be handled by ordinary unification in this way. Luckily the pretty printing routines will turn such complex turns back into something more readible:

```
| ?- X: agr => (sg & mas ; ~2 & neut), show(fs,latex,[value(X)]).
```

12.1. Hook Predicates

This section lists the hook predicates used by the hdrug_feature library.

12.1.1. top(Subtypes)

Defines all sub-types of top as a list of atoms.

12.1.2. type(Type,Subtypes,Attributes)

Defines a Type with Subtypes and Attributes. In general, Subtypes is a list of list of types. If a list of types [T0..Tn] is given, then this is automatically converted to [[T0..Tn]].

12.1.3. at(Type)

Type is an atomic type, i.e. without any sub-types and without any attributes.

12.1.4. list_type(Head,Tail)

Declares Head and Tail to be the attributes to refer to the head and the tail of objects of type ‘list’.

12.1.5. extensional(Type)

Declares Type to be an extensional type, i.e. no extra variable is added to objects of this type; extensional objects are identical if they have the same value for each of their attributes. Intensional objects are identical only if they have been unified.

12.1.6. boolean_type(Type,Model)

Declares Type to be a boolean type with Model as its model (list of list of atoms). For instance, boolean_type(agr, [[1,2,3], [sg,pl], [mas,fem,neut]]) defines that agr is such a boolean type.

12.1.7. intensional(Type)

Type must be a boolean type. Boolean types are extensional by default, unless this predicate is defined for them.

12.2. Predicates

This section lists the predicates exported by the hdrug_feature library.

12.2.1. hdrug_feature:pretty_type(Type)

pretty prints information on Type. Types should have been compiled with hdrug_feature:type_compiler.

12.2.2. hdrug_feature:find_type(?Term,-Types[,-Atts])

Types will be bound to the list of most informatives sub-types of Term; Atts will be bound to the list of all attributes of Term. Meta-logical. Types should have been compiled with hdrug_feature:type_compiler.

12.2.3. hdrug_feature:unify_except(T1,T2,Path)

T1 and T2 are Prolog terms. Path is a sequence of attributes separated by colons. The predicate evaluates T1:Path and T2:Path (in order to ensure that Path is consistent with both objects. Furthermore, T1 and T2 are unified except for the values at T1:Path and T2:Path. Types should have been compiled with hdrug_feature:type_compiler.

12.2.4. hdrug_feature:unify_except_l(T1,T2,ListOfPaths)

Similar to unify_except, except that the third argument now is a list of paths. T1 and T2 are Prolog terms. Each path in ListOfPaths is a sequence of attributes separated by colons. The predicate evaluates for each Path, T1:Path and T2:Path (in order to ensure that Path is consistent with both objects. Furthermore, T1 and T2 are unified except for all values at T1:Path and T2:Path for Path in ListOfPaths. Types should have been compiled with

hdrug_feature:type_compiler.

12.2.5. hdrug_feature:overwrite(T1,T2,Path,Type)

Abbreviation for unify_except(T1,T2,Path), T2:Path => Type; i.e. T1 and T2 are identical, except that T2:Path is of type Type. Types should have been compiled with hdrug_feature:type_compiler.

12.2.6. hdrug_feature:(ObjPath => Type)

This predicate evaluates ObjPath, and assigns Type to the result (i.e. the result is unified with the Prolog term representation of Type). ObjPath is a Prolog term followed by a (possibly empty) list of attributes separated by the colon :. A path such as X:syn:head:cat refers to the cat attribute of the head attribute of the syn attribute of X. Type must be a type (Prolog atom) or a boolean expression of boolean types. Types should have been compiled with hdrug_feature:type_compiler.

12.2.7. hdrug_feature:(ObjPath /=> Type)

This predicate evaluates ObjPath, and ensures that it is not of type Type (i.e. the result is not allowed to subsume the Prolog term representation of Type). ObjPath is a Prolog term followed by a (possibly empty) list of attributes separated by the colon :. A path such as X:syn:head:cat refers to the cat attribute of the head attribute of the syn attribute of X. Type must be a type (Prolog atom) or a boolean expression of boolean types. Types should have been compiled with hdrug_feature:type_compiler. The implementation of this construct uses delayed evaluation.

12.2.8. hdrug_feature:(ObjPath ==> Term)

This predicate evaluates ObjPath, and unifies Term with the result. ObjPath is a Prolog term followed by a (possibly empty) list of attributes separated by the colon :. A path such as X:syn:head:cat refers to the cat attribute of the head attribute of the syn attribute of X. Term is an arbitrary Prolog term. This predicate is often used to include arbitrary Prolog terms inside feature structures. You can define a hook predicate catch_print_error/3 in order to define pretty printing for such terms. Types should have been compiled with hdrug_feature:type_compiler.

12.2.9. hdrug_feature:(ObjPathA <=> ObjPathB)

This predicate evaluates PathA and PathB, and unifies the results. ObjPathA and ObjPathB each is a Prolog term followed by a (possibly empty) list of attributes separated by the colon :. A path such as X:syn:head:cat refers to the cat attribute of the head attribute of the syn attribute of X. Types should have been compiled with hdrug_feature:type_compiler.

12.2.10. hdrug_feature:(PathA <?=?> PathB)

This predicate uses the if_defined/2 construct in order to unify two paths, provided each of the two paths is defined. It is defined by:

```
A <?=?> B :-
    if_defined(A,Val),
    if_defined(B,Val).
```

12.2.11. hdrug_feature:is_defined(Path,Bool)

This predicate evaluates Path. If this is possible (i.e. the attributes are all appropriate) then Bool=yes. Otherwise Bool=no.

12.2.12. hdrug_feature:if_defined(Path,Val[,Default])

This predicate evaluates Path, and unifies the result with Val. If the path cannot be evaluated (for instance because a feature is used which is not appropriate for the given type) then the predicate succeeds (in the binary case) or unifies Val with Default (in the ternary case). For example:

```
if_defined(X:head:subcat,List,[]),
```

could be used as part of the definition of a valence principle, in order to obtain the list value of the subcat attribute. However, for categories which have no subcat attribute, List is instantiated to [].

12.2.13. hdrug_feature:type_compiler[(Module)]

Compiles type declarations (loaded in Module or user) into definitions for the predicates =>/2, <=>/2, ==>/2, unify_except/3, overwrite/4. The type declarations consist of definitions for the hook predicates top/1, at/1, type/3, list_type/2, extensional/1, boolean_type/2, intensional/1. The top/1 declaration is required.

top(Subtypes) is an abbreviation for type(top,[Subtypes],[]).

at(Type) is an abbreviation for type(Type,[],[]).

type(Type,[T0,..,Tn],Atts), where each Ti is atomic, is an abbrevation for type(Type,[[T0,..,Tn]],Atts).

Each type is specified by a list (conjunction) of lists (exclusive disjunctions) of subtypes and a list of attributes.

Objects of type type(Type,[[A1..An],[B1..Bn],...,[Z1..Zn]],[Att1..Attn]) will be represented by the Prolog term Type(Ai',Bi',..,Zi',Att1',..,Attn',_)

For example, the declaration

```
type(sign,[[basic,complex],[nominal,verbal]],[mor,sem])
```

implies that everything of type sign is represented with a term sign(BorC,NorV,Mor,Sem,_) where the first argument represents the first sub-type (basic or complex and any associated information with these subtypes), the second argument represents the second subtype (nominal or verbal), the third argument represents the value of the ‘mor’ attribute, and the fourth argument represents the value of the ‘sem’ attribute. The fifth argument is introduced in order that such objects are ‘intensional’: objects are identical only if they have been unified.

Assumptions:

‘top’ has no appropriate features, will always be denoted with Variable bottom has no appropriate features, will not be denoted -> failure hence top is only specified along one ‘dimension’ (use top/1).

Other types can be further specified along several dimensions, hence can have more than one subtype, at the same time. Subtypes of a type are mutually exclusive (in the example above, you cannot be both nominal and verbal).

All types describe intensional objects (as in PATR II). For this purpose, during compilation an extra argument position is added to which you cannot refer. You can use extensional/1 for a specific type in order that this extra position is not added.

Boolean types.

The technique discussed in Chris Mellish’ paper in Computational Linguistics is available to be able to express boolean combinations of simple types. First, boolean types are declared using the hook predicate boolean_type(Type,ListOfLists). For example, the declaration

```
boolean_type(agr,[[1,2,3],[sg,pl],[mas,fem,neut]])
```

declares that objects of type ‘agr’ are elements of the cross-product of {1,2,3} x {sg,pl} x {mas,fem,neut}. Instead of simple types, boolean combinations are now allows, using the operators & for conjunction, ~ for negation and ; for disjunction.

```
?- X => (sg & ~fem ; pl).

X = agr(0,_A,_B,_C,_C,_D,_E,_F,_G,_H,_H,_I,_J,_K,_L,_M,_M,_N,1) ?
```

13. hdrug_show: Visualization

The libraries contain predicates to visualize trees, feature-structures and Prolog terms (including Prolog clauses). A number of different output media are available: LaTeX, Tcl/Tk, CLiG, DOT, and ordinary text output. The visualization tools are all available by means of a single generic predicate show/3.

Viewing Prolog Terms representing Feature Structures

Note that a couple of predicates are available to view Prolog terms as feature structures. Again, the predicate show/3 is available as an interface to this functionality. For example, you might try the conjunction:

```
Y:cat:agr <=> Y:cat:subj:cat:agr, show(fs,tk,[value(Y)]).
```

Instead of **tk**, any of the identifiers **latex**, **user**, **clig**, **dot** can be used to direct the output to a different medium. For instance, the query

```
Y:cat:agr <=> Y:cat:subj:cat:agr, show(fs,latex,[value(Y)]).
```

But if you insist on ordinary output, try:

```
show(fs,user,[value(X)]).
```

This produces:

```
{sign}
|cat {verb}
|   |agr <A>
|   |subj {sign}
|   |    |cat {cat}
|   |    |   |agr <A>.
```

Not only can you view feature structures this way, but also clauses; cf. show/3 below.

Tree Formats

The libraries contain extensive possibilities to produce output in the form of trees. Only a few declarations are needed to define what things you want to see in the tree. In effect, such declarations define a ‘tree format’.

In Hdrug, there can be any number of tree formats. These tree formats are named by a ground identifier. A tree format consists of three parts: the **path** definition indicates what part of the object you want to view as a tree; the **label** definition indicates how you want to print the node of a tree; and the **daughter** definition indicates what you consider the daughters of a node.

Because we want to be able to have multiple tree formats around, we must declare the corresponding predicates ‘multifile’, as otherwise existing tree formats would be erased.

For example, the following predicates define a tree-format called ‘s’ (this example is taken from the ‘Dcg’ application).

```
:- multifile graphic_path/3.
graphic_path(s,node(_,S),S).

:- multifile graphic_label/3.
graphic_label(s,Term,Label) :-
    functor(Term,Label,_).

:- multifile graphic_daughter/4.
graphic_daughter(s,1,Term,D) :-
    arg(1,Term,D).

graphic_daughter(s,2,Term,D) :-
    arg(2,Term,D).
```

The first predicate defines that we want to take the semantics part of a node as the term that we want to view as a tree. The second predicate defines that for a given tree Term we want to print its functor as the node label. Finally the third predicate defines that for a given tree Term the first daughter is to be the first argument of the term, and the second daughter is to be the second argument.

As another example of a tree format definition, consider the constraint-based Categorial Grammar application. application. Here we find:

```
:- multifile graphic_path/3.
graphic_path(syn,Obj,Obj).

:- multifile graphic_label/3.
graphic_label(syn,tree(Sign,_,[_|_]),Label) :-
    cat_symbol(Sign,Label).

graphic_label(syn,tree(W,_,[]),W).

:- multifile graphic_daughter/4.
graphic_daughter(syn,No,tree(_,_,[H|T]),D) :-
    nth(No,[H|T],D).
```

Here, objects generally are of the form tree(Node,_,ListOfDs). Therefore, the path part of the tree format definition simply unifies the object and the tree part. The label part of the tree format definition distinguishes two cases. If there are no more daughters, then the node is a terminal, and this terminal is simply taken to be the node label. In the other case the node label is defined by a seperate predicate ‘cat_symbol’. This predicate changes the internal representation into some more readable format. Finally, the daughter part of the tree format definition uses the Sicstus library predicate ‘nth’. The effect of the definition is that the first daughter is the first element of the daughter list, etc.

Tk Output

The library defines the predicate show/3 index{show (predicate)} as a generic interface to the visualization tools. If a tree is to be displayed on the Tcl/Tk Canvas widget, then we can use this predicate by taking the desired tree format as the first argument, the atom { t tk} as the second argument, and a list of objects we want to be displayed as the third and final argument. For instance:

```
?- findall(object(A,B), object(A,B), Objects),
show(syn,tk,Objects).
```

If the tree is output thru the Tk/Tcl canvas, then the nodes of the trees are buttons. For each tree format we can define what action should be undertaken if a button is pressed. This is defined by the predicate show_node/2. The first argument is the identifier of the tree format, the second argument is the current node (note: this is not the label as defined by graphic_label, but the term on the basis of which graphic_label is defined).

The following definition, from the Constraint-based Categorial Grammar application, prints the node as a feature structure in a separate Tk window.

```
show_node(syn,tree(Sign,[_|_],_)) :-
        show(fs,tk,[value(Sign)]).
```

If this predicate is not defined then the label will simply be written out as a Prolog term to standard output.

Similarly, the predicates show_node2/2 and show_node3/2 can be used to define an action for pressing the second and third mouse-button respectively. Generally these predicates should be defined multifile.

LaTeX output

The predicate show/3 is also used to produce LaTeX output of trees. A variant of the previous example produces LaTeX:

```
?- findall(object(A,B), object(A,B), Objects),
show(syn,latex,Objects).
```

This ensures that a LaTeX file is created and the appropriate shell commands are called to get ghostview to display the tree. The first argument is the name of a tree-format.

CLiG Output

A further possibility concerns is to use the CLiG system for displaying output. In that case the example becomes;

```
?- findall(object(A,B), object(A,B), Objects),
show(syn,latex,Objects).
```

Dot Output

For trees, you can also use the DOT graph visualization programme.

ASCII Art Output

Ordinary text (to standard output) is available as well; in that case the identifier **user** is used:

```
?- findall(object(A,B), object(A,B), Objects),
show(syn,user,Objects).
```

Trees of feature structures

Trees in which each of the nodes is a feature-structure are supported for Tk output and LaTeX output. Nodes are interpreted as a description of a feature-structure if the tree format identifier matches matrix(_).

User defined action for a given node can be obtained using a tree format which matches user(_). In such a case you are responsible for displaying a given node by defining the predicate tk_tree_user_node/2 where the first argument is the label of the current node, and the second argument is a Tcl/Tk frame identifier already packed as part of the tree, which can be further worked upon.

Visualization of clauses

The third argument of the predicate show can be a clause. An is example is `http://www.let.rug.nl/~vannoord/Hdrug/Manual/clause.png`

Visualization of the type declarations

Refer to the predicates pretty_type/0, pretty_type/1.

14. help: The Help System

The help module provides support to create both on-line and off-line documentation on Prolog programs. Documentation must be defined by the hook predicate help_info/4. Documentation on a per module basis is provided if a help_info(module,Module,TitleString,DescriptionString) definition is given for Module. In that case the system also checks for Module:help_info/4 definitions.

The module supports production of the help information on standard output, (which can be converted into html format), and there also is an interface to a graphical user interface based on library(tcltk).

14.1. List of Hook Predicates

This section lists the hook predicates which an application can define for the help module.

14.1.1. help_info(Class,Key,Usage,Expl)

Provides help information for Class and Key (both must be atoms). Usage and Expl are Prolog strings. Typically the Usage string is a short summary, and Expl is a longer explanation. Class is typically pred, hook, flag, command, option, etc. Note that each module can have its own help_info predicates. You can also define user:help_info/4 declarations on the special class module. In that case, if a full documentation on a module is requested the Usage string is used as the title and the Expl string as an introduction to the module. There can also be Module:help_info/4 declarations on the special class ‘class’. If a full listing on a class in Module is requested, then Usage and Expl are used as the title and introduction to that section.

14.2. List of Predicates

This section lists the predicates defined by the help module.

14.2.1. help_listing

Lists all help information.

14.2.2. help/help(Module)/help(Module,Class)

Use help/0 to see for which modules help is available. Use help/1 for an overview which classes are available for a given module. Use help(Module,Class) to see for which keys help is available.

14.2.3. help_module[(M)]

Use help_module(M) for a full listing of the help information available on module M. Without M uses module user.

14.2.4. help_class(C[,M])

Use help_class(C,M) for a full listing of the help information available for class C in module M. Without M module user is assumed.

14.2.5. help_key(K[,C[,M]])

Use help_key(K,C,M) for a full listing of the help information available for key K in class C in module M. If C (and M) are not given, then use variable for C (and M).

14.2.6. help_add_to_menu(Menu,Interp)

Interface of the help system and a graphical user interface based on library(tcltk). Menu must be a menu already existing for Tcl/Tk interpreter Interp. The various help messages are added as cascaded menu entries in Menu. Cf. also the help/1 predicate and the help_info/4 hook predicate.