Projekt PAPA
Arbeitsbereich NATS
Fachbereich Informatik
Universität Hamburg
Memo HH-XX
30 August 2003
Kilian Foth, Stefan Hamerich,
Ingo Schröder, Michael Schulz,
Tomas By

[X]CDG User guide

Version 1.3

Documentation for the CDG system
including the graphical interface XCDG

Contents

Chapter 1

The cdg parser

This chapter describes the use of the CDG (constraint dependency grammar) parsing system. It is based on the DAWAI reports HH-1 to HH-4 (Schröder, 1997c; Schröder, 1997a; Schröder, 1997d; Schröder, 1997b).

The parsing theory that the software is based on is *not* explained here; it is assumed the reader is familiar with it. Maruyama (1990a), Maruyama (1990b), Maruyama, Watanabe and Ogino (1990), Menzel (1994), Menzel (1995), Harper et al. (1993), Harper et al. (1994), Harper and Helzerman (1994), Schröder (1995) as well as Schröder (1996) and particularly Heinecke et al. (1998) describes comprehensively parsing by constraint analysis. The precise syntax of the inputs to the system is described in section 1.4.

1.1 Command line format

The program 'cdg' is interactive and supports only a few command line options.

At present the following options are available:

- The option '-q' turns off additional messages that are not required in non-interactive mode. This option corresponds to the interactive command 'set verbosity off'.
- The option '-d' turns on the display of deleted values in the output of the 'net' command. This option corresponds to the interactive command 'set showdeleted on'.
- The option '-s' switches on the use of statistical parameters *Warning:* This is not officially supported, nor further documented.
- The option '-e' prevents the initialization of edges during the production of constraint nets, to increase the speed. *Warning:* This option should only be used if the search is started with the command 'netsearch'. Otherwise the behaviour is undefined.[1]

[1] The options '-s' and '-e' were tailored for a specific application and have not generally been tested under other conditions.

- The option '-m' changes the behavior of the system during search in constraint networks: when one solutions is found, all level values in the net that are not part of the solution are deleted; if several solutions are found, then all level values that are not part of any solution are deleted. This is indicated by the status line `search modifies net: yes`.
- The option '-n' corresponds to the command `set normalize on`.
- The option '-i name' sets the name of the initialization file to 'name' (default name: `.cdgrc`).
- The option '-x' corresponds to the command `set xml on`.

At start-up, the program first loads the initialization file, whose name is normally `.cdgrc` but can be changed with the option '-i'. The file is searched for in the current directory, and then the user's home directory. If the initialization file is found, then its contents are executed, line by line, as if they had been entered interactively.

Then all files whose names were given on the command line are loaded. The following example illustrates the command:

```
% cdg test/menzel

CDG parser
Ingo Schröder ingo.schroeder@informatik.uni-hamburg.de
Type 'help' for help.

file 'test/menzel' loaded: 12/2/8/10/0/0/0
cdgp>
```

The numbers after the message that the file was loaded, indicate, from left to right, the numbers of constraints, level definitions, lexicon entries, word graphs, annotations, hierarchy definitions, and the statistical parameters.

1.2 Interactive commands

The available commands are listed in the table on page 6 and described below in alphabetical order.

1.2.1 The command 'activate'

The command 'activate' (re-)activates the specified constraint classes, so that they are used in subsequent computations. If no parameters are given, all known classes are activated.

See also the command 'deactivate'.

```
cdgp> activate map
```

Input	`load`	Load files
	`inputwordgraph`	Specify word graph
	`newnet`	Create constraint net from word graph
	`renewnet`	Reset constraint nets
	`netdelete`	Delete constraint nets
	`parsedelete`	Delete dependency analyses
	`reset`	Reset the system
Actions	`netsearch`	Search for solutions of constraint net
	`incrementalcompletion`	Search for dependency analyses of word graph
	`frobbing`	Heuristic search for solutions of c. net
	`gls`	Transformation-based search for solutions of c. net
Settings	`set`	Control various parameters
	`net`	Toggle use of specified constraint net
	`useconstraint`	Toggle use of specified constraints
	`activate`	Activate constraint classes
	`deactivate`	Deactivate constraint class
	`weight`	Change the weights of constraints
	`uselevel`	Toggle use of specified levels
	`levelsort`	Set level sort order
Output	`compareparses`	Compare two dependency analyses
	`verify`	Compare dependency analyses to annotations
	`printparse`	Print one dependency analysis
	`printparses`	Print all dependency analyses for one c. net
Nice output	`writewordgraph`	Print word graphs
	`writenet`	Print constraint net
	`writeparses`	Print all dependency analyses for one c. net
	`writeannotation`	Write best parse of a net to disk
	`annos2prolog`	Write all annotations to a file in Prolog format
	`parses2prolog`	Write all parses to a file in Prolog format
Diagnostics	`constraint`	Show constraints
	`lexicon`	Show lexicon entries
	`wordgraph`	Show word graphs
	`annotation`	Show annotations
	`hierarchy`	Show hierarchies
	`level`	Show levels
	`section`	Show constraint classes
	`anno2parse`	
	`status`	Show information about the system
	`edges`	List edges in constraint net
	`distance`	Show distances in a constraint net
	`nonspeccompatible`	
	`hook`	Control certain output
	`showlevel`	Toggle display of levels
Program Control	`quit`	Terminate

1.2.2 The command 'anno2parse'

TBD

1.2.3 The command 'annos2prolog'

Writes all currently loaded annotations to the specified file in Prolog format as per the following (pseudo-)BNF. The number of annotations that were written to the file are printed on the screen.

```
<File>        ::= { <Annotation> }*

<Annotation>  ::= annotation( <Id> , <LaId> , <Words> ).

<Words>       ::= <Prolog list of Word>

<Word>        ::= word( <From> , <To> , <String> , <Specs> )

<From>,<To>   ::= <Prolog integers>

<Id>,<LaId>,<String> ::= <Quoted Prolog atoms>

<Specs>       ::= <Prolog list of Spec>

<Spec>        ::= tag( <Attribute>, <Value> )
                |  dep( <Level>, <Label>, <Position> )

<Attribute>,<Value>,<Level>,<Label> ::= <Quoted Prolog atoms>

<Position>    ::= <Prolog integer, starting at 1>
```

The lattice label (`LaId`) identifies the word graph, as in the output from `parses2prolog` (section 1.2.33). There will always be exactly one `dep`/3 term per level in each `annotation`/3 structure.

1.2.4 The command 'annotation'

The command 'annotation' lists loaded analyses. With no parameter, all loaded analyses are displayed. Any parameters to the command are interpreted as identification of the analyses to be shown.

```
cdgp> annotation n001k/000/2
n001k/000/2 :
'ja prima dann lassen Sie uns doch noch einen Termin ausmachen
wann waere es Ihnen denn recht' :
wann:
  cat/PWAV
  syn->AMOD->2
waere:
```

```
  cat/VFIN
  syn->VFIN->0
es:
  cat/PPERIR
  syn->SUBJ->2
ihnen:
  cat/PPERIR
  syn->OBJD->2
denn:
  cat/ADV
  syn->AMOD->2
recht:
  cat/ADJD
  syn->PRED->2
```

1.2.5 The command 'chunk'

Exercises the chunker defined by 'chunkerCommand' on the specified wordgraph and prints out the result.

1.2.6 The command 'compareparses'

The command 'compareparses' compares two dependency analyses and reports detailed information about the differences. The two analyses are specified by name. If the second name is missing, the dependency analysis that was produced last is used as point for comparison.

1.2.7 The command 'compile'

The command 'compile' translates the loaded Constraint grammar into machine code, for efficiency. *This command has not been implemented yet.*

1.2.8 The command 'constraint'

The command 'constraint ' lists the loaded constraints. The parameters indicate which constraints to list. If no parameters are given, all constraints are listed.

```
cdgp> constraint se2
{ X } : se2 : sem : 0.700000 :
(((X.level = SEM & X^word = fressen) & X@prop = animal) -> X.label = AG);
```

1.2.9 The command 'dbclose'

This command cancels the effect of 'uselexicon'. **Note:** Lexicon items that were already loaded from disk remain in memory. Only future queries are affected.

1.2.10 The command 'deactivate'

The command 'deactivate' takes a constraint class as parameter. Members of that class are not used in subsequent computations. If no parameters are given, all known classes are deactivated.

See also the command 'activate'.

```
cdgp> deactivate map
```

1.2.11 The command 'distance'

The command 'distance' shows a matrix of the approximate distances between the word forms in a constraint net.

```
cdgp> distance net0
          die  Frau   sieht    die Tochte

   die         +0001  +0002  +0003  +0004
  Frau  -0001         +0001  +0002  +0003
 sieht  -0002  -0001         +0001  +0002
   die  -0003  -0002  -0001         +0001
Tochte  -0004  -0003  -0002  -0001
```

1.2.12 The command 'edges'

The command 'edges' lists some or all of the edges in a constraint net. It takes one non-optional parameter, identifying the constraint net, and two optional parameters, specifying the vertex indexes which are shown, for example, by the command 'net'.

```
cdgp> edges net0 0 4
pferde(0,1)-SYN  --->  gras(2,3)-SYN
start v    stop >|  SUBJ/fressen      OBJ/fressen
-------------------------------------------------
 SUBJ/fressen    |   0.000000e+00     3.000000e-01
  OBJ/fressen    |   1.000000e-01     0.000000e+00
```

1.2.13 The command 'frobbing'

The command 'frobbing ' starts a heuristic search for solutions in a constraint net by transformation of incorrect value assignments. This procedure sacrifices completeness to gain speed, and it is not guaranteed that a solution is found.

The parameters are `key=value` pairs with the following meaning:

Key	Meaning	Default value
`agenda <num>`	agenda size of local search	`3000`
`batch <yes|no>`	skip interactive part	`no`
`beam <num>`	beam width for local evaluation	unlimited
`execute <string>`	commands for `manual`	""
`freeze <yes|no>`	freeze results of phase 1?	yes
`method <name>`	frobbing method	`combined`
`maxnsize <name>`	maximal subproblem size	30
`minsize <name>`	minimal subproblem size	6
`net <name>`	net to search	newest constraint net
`parse <name>`	parse to frob	newest parse
`pressure <num>`	enforced minimum score	0.9
`strict <yes|no>`	isolate subproblems totally?	yes
`subproblem <name>`	fragmentation method	""

Specifying the search method explicitly is not very useful, since the default method is nearly always the best. The other methods are experimental (Foth, 1999).

The method `manual` lets the user transform dependency analyses interactively. Usually (unless `batch` is used) this mode is entered after the time limitation of another search method has been reached. The command '`q`' terminates interactive processing and returns the control to the main program. The command '`h`' gives an overview of commands available during manual frobbing.

1.2.14 The command 'gls'

This command applies the GLS solution procedure to a constraint net. GLS is transformation-based and uses local search with hill climbing to avoid extreme points in the search space when converging. As in frobbing (section 1.2.13) dependency structures are transformed if constraints are violated. While frobbing is based on taboo search (Glover, 1989; Glover, 1990), GLS uses a weight-based control heuristic. Characteristics of extreme points in the weights are stored and this information is used to compute the weighing function of the PCSPs. This lets the local search break out of extreme points and continue the search in more promising regions of the search space.

In the current implementation, GLS allows several different GLS heuristics to scan the search space independently. This procedure is called MGLS (multiple steered local search) and assigns each parameterized heuristic to an agent using its own 'GLS search net.' The set of all agents of a MGLS is called the 'GLS pool.'

Unfortunately, real concurrency has not yet been implemented, and the system simulates concurrency in its own run time environment by giving individual agents time slices according to the round-robin principle.

The fundamental algorithms used here are described in (Voudouris, 1997). A detailed description of the conversion and extensions can be found in (Schulz, 2000).

The command '`gls`' has the following syntax:

```
cdgp> gls -?
INFO: gls [<netId>] [<options>]*
      <netId>                          : name of constraint net
      -c, --cutoff [<x>]+              : maximum costs of one hard violation
```

```
-co, --costs [<x>]+               : maximum allowed augmented costs
-coop, --cooperate                : networks learn from each other
-comp, --compete                  : networks don't learn from each other
-cmp, --compare <name>            : compare function 'badness' or 'costs'
-prio, --priority [<n>]+          : cycles spent on a network per timeslice
-d, --debug <n>                   : debug level
-dt, --detour [<n>]+              : maximum allowed cycles to spend on worse bindings
-from, --from-cycle [<n>]+        : start computing at the specified cycle
-fup, --force-pruning [<n>]+      : detour length after which pruning is forced
-i, --interrupt [<n>]             : interrupt gls after given number of cycles
-iup, --initial-unary-pruning     : pruning fraction in init phase
-g1, --guide1 [<x>]+              : strength of penalization
-g2, --guide2 [<x>]+              : strength of reinforcement
-pup, --prolong-unary-pruning     : pruning fraction in prolongation phase
-n, --normalize [<x>]+            : normalization factor
-s, --statistics <filename>       : generate statistics and print them into a file
-t, --time [<n>]+                 : maximum milliseconds available to solve the problem
-tup, --termination-unary-pruning : pruning fraction in termination phase
-to, --to-cycle [<n>]+            : stop computing at the specified cycle
-tol, --tolerance [<x>]+          : tolerated utility of a binding to be repaired
-u, --utility [<x>]+              : minimal utility of a binding to be repaired
-up, --unary-pruning [<x>]+       : pruning fraction applied in any phase
-v, --version                     : show version of gls
-?, -h, --help                    : this text

<n>                               : integer
<x>                               : float
[<p_i>]+                          : iterated parameters; parameters at position i are
                                    assigned to net i; if there are more parameters
                                    than nets then new nets are created with default
                                    parameters set to the most previously created net
                                    in the pool
```

The parameters of the command can be divided into five categories:

1. Selection function parameters:
   (a) Threshold value `-cutoff`
   (b) Normalization `-n`
   (c) Penalty weight `-g1`
   (d) Reinforcement weight `-g2`
   (e) Tolerance `-tol`
   (f) Comparison function `-cmp`
2. Combined control heuristics
   (a) Simple steered local search
   (b) Multiple steered local search
      i. Cooperative, parallel `-coop`
      ii. Competing, parallel `-comp`
      iii. Serial `-from`, `to`
      iv. Prioritized `-prio`
3. Termination criteria

(a) Maximum combined cost `-co`

(b) Minimal usefulness `-u`

(c) Time `-t`

(d) Cycles `from`, `to`

4. Extended heuristics

   (a) Unary pruning `-up`

      i. Unary pruning in the initialization phase

      ii. Unary pruning in the prolongation phase

      iii. Unary pruning in the termination phase

   (b) Limitation of the transformation paths `-dt`

5. Debugging and statistics

   (a) Level `-d`

   (b) GLS command mode `-i`

   (c) Generation of statistical data `-s`

Examples

Here follows some examples of the different parameters. Assume a constraint net `net0` was produced from a word graph using the `newnet` command (see section 1.2.30). If `net0` were the last constraint net produced, then the net identifier (`<netid>`) can be omitted from the 'gls' command.

a) Default parameters and GLS agent:

```
cdgp> gls
INFO: renewing net 'net0'
PROFILE: it took 170ms to establish the pool

  ------------------------------------------------------------------------------
  settings:
  ------------------------------------------------------------------------------
  net                         : net0
  wordgraph                   : n001k001-1
  debug level                 : 1
  compare function            : badness
  interruption                : no
  parallelity                 : single
  ------------------------------------------------------------------------------
                                    0
  ------------------------------------------------------------------------------
  cutoff                 :  1.000e+01
  penalty guide          :  1.000e+00
  reinforcement guide    :  0.000e+00
  normalization(beta)    :  1.000e+00
  pruning fraction       :  0%/0%/0%
  force unary pruning    :          0
  priority               :          1
  from cycle             :          0
  to cycle               :  (unbound)
```

```
  max time               :   1 m 40 s
  max detour             :  (unbound)
  max costs              :       1000
  min utility            :  (unbound)
  tolerance              :  0 %

  ------------------------------------------------------------------------------

* 0 :     2 :   260ms : util=3.845e-01/1.000e+04/3.845e-01 : score= 4/20/2.112e-05 ...
                                                          ... : costs= 94.96/ 94.96/135.53
* 0 :    13 :   400ms : util=3.998e-01/1.000e+04/5.036e-01 : score= 4/20/7.943e-05 ...
                                                          ... : costs= 83.70/ 83.70/178.61
* 0 :    16 :   490ms : util=4.158e-01/1.000e+04/5.357e-01 : score= 3/20/1.105e-07 ...
                                                          ... : costs=103.73/104.73/178.61
* 0 :    17 :   580ms : util=4.616e-01/1.000e+04/5.357e-01 : score= 3/20/1.920e-05 ...
                                                          ... : costs= 85.00/ 86.00/178.61
INFO: net 0 finished initialization
* 0 :    22 :   680ms : util=4.849e-01/4.849e-01/5.357e-01 : score= 1/26/5.270e-08 ...
                                                          ... : costs= 94.20/ 94.20/178.61
* 0 :    42 :   830ms : util=6.191e-01/4.791e-01/6.191e-01 : score= 0/24/8.651e-06 ...
                                                          ... : costs= 56.21/ 56.21/178.61
INFO: net 0 finished prolongation
* 0 :    59 :   980ms : util=4.504e-01/4.504e-01/6.191e-01 : score= 0/17/3.757e-01 ...
                                                          ... : costs=  9.50/ 13.50/178.61
  0 :  2382 :  1820ms : util=7.594e-01/4.504e-01/8.366e-01 : score= 0/17/3.757e-01 ...
                                                          ... : costs=  9.50/924.50/977.76
INFO: further usage of net 0 too expensive
INFO: best parse found is 'parse7' with score 3.757e-01

  statistics:
  ----------------------------
  total          :      1840 ms       2437 cycles        2990 repairs   2438 penalties
  solution       :       980 ms         59 cycles       53.26 %        3.757e-01 score
  soft solution  :       830 ms         42 cycles       45.11 %        8.651e-06 score
  benchmark      :      1324 c/sec    1625 r/sec           1 r/c
  blind alleys   :         0
  detour         :      2379 max         20 needed
  costs          :   1005.76 max     178.61 needed
  utility        : 8.366e-01 max  4.504e-01 needed   4.504e-01 min
```

The first part of the output above (starting with `settings:...`), the current parameters of the GLS are listed in two tables: one for parameters common to the GLS pool, and one with one column per GLS agent, showing agent specific parameters.

Up to five different debug levels, with increasing verbosity, can be selected with the option `-d <n>`. The comparison function is indicated by '`compare function: badness`,' and can be selected with the option `-cmp <name>`. There are two possible functions `badness` and `costs` for comparing weights in the dependency analysis. The default value is `badness`. The weighing function compares each local extreme, into which the search converges, with the best previous dependency analysis in the GLS pool. The line '`interruption: no`' indicates that the algorithm has not been interrupted. During analysis it is possible to interrupt and go into GLS command mode by pressing `Ctrl-C`. The option `-i <n>` makes the system stop and enter GLS command mode automatically after the specified number of cycles. The line `parallelity : single` describes the type of the GLS pool. If there is more than one GLS agent, the method for combining the heuristics can be selected with the options `-coop` and `-comp` (see examples d) and e)). If neither is specified the agents co-operate. For now, it is only possible for all GLS agents to either co-operate or compete.

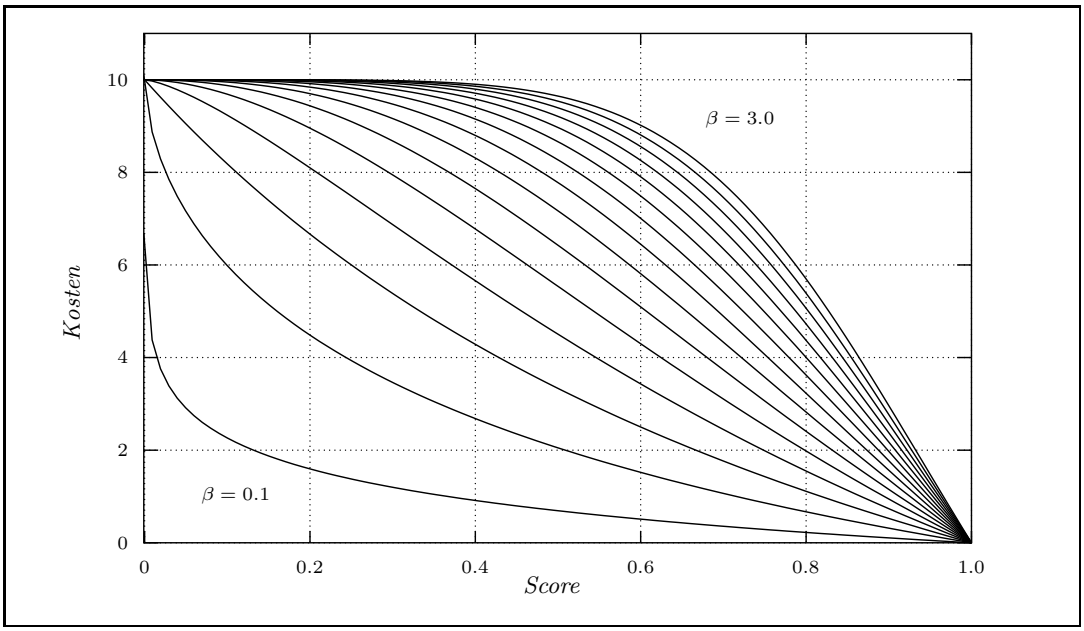


Figure 1.1: Conversion of scores into costs

In the second part of the '`settings`' the parameters of each GLS agent is shown in a separate column. All GLS agent parameters are listed, with the initial values. The effect of the parameters `cutoff` and `normalization` on the conversion of scores into costs is shown in figure 1.1, and can be written mathematically as follows.

$$costs = |tanh(log(score) \cdot \beta)| \cdot \gamma$$

The normalization factor is β, and γ is the cutoff. The normalization factor influences the heuristic selection of violated constraints to be repaired in each cycle. If $\beta \leq 1$, the search will be concentrated to areas of the search space with strong constraint violations, since large addends of the costs of an injury are to due rather to individual strong Constraints.

If several weak constraints are violated, the normalization will decrease the combined weight. Setting $\beta \geq 1$ has opposite effect: weak constraints are satisfied first, until only strong violations remain. To sum up, an agent with a small normalization factor will quickly obtain a solution with no hard constraint violations, but will find it harder to remove the remaining soft violations; with a large normalization factor it is the other way around. Values $0.4 \leq \beta \leq 0.7$ givers a good average of both these effects.

Other important parameters that control a GLS agent, are `-g1` (λ_1) and `-g2` (λ_2). They adjust the influence of the weighted terms of the augmented cost function, which has the following form:

$$costs_{aug} = costs + \lambda_1 \cdot w_1 - \lambda_2 \cdot w_2$$

The GLS heuristics increases the weight w_1 to lower the preference for the associated dependency structures.[2] If a dependency structure is used, which is in a local extreme

[2] At present only individual dependency edges have weights.

point into which the search converges, then weight w_2 is increased, in order to heighten the preference for the used structures.

Time limitations can be set for each agent. The start and end cycle within which an agent is active can be specified with `-from` and `-to`; the maximum time that the agent has at its disposal with `-t`. The option `-tt` specifies the maximum augmented costs of a dependency analysis, and the option `-u` limits the minimum usefulness granted to an agent. By default GLS uses a maximum time of 100000 milliseconds. The default value of maximal augmented cost is to 1000. Reasonable values are grammar dependent in each case and should always be determined *by hand*. All other parameters of a GLS agent are for the time being unused.

The output produced by GLS during transformation can be configured with the system-wide switch '`progress`' (see section 1.2.41). A propeller-like animation and statistics about the current search condition can be produced, and extreme points in the search space are indicated with a star (`*`) at the beginning of the status line, which has the following parts:

(a) Marking or animation at the start of line; Stars (`*`) indicate an improved dependency analysis.

(b) Identification of the active GLS agent.

(c) Time used by the algorithm.

(d) Usefulness of the active agents, divided into current, minimal, and maximal usefulness (found so far).

(e) Quality of the search state, divided into number of hard and weak conditions violated, and the score without considering hard violations.

If an agent terminates, the reason is reported. In the example above, the line

```
INFO: further usage of net 0 too expensive
```

says that the augmented cost of the current search state exceeds the limit. The line

```
INFO: best parse found is 'parse7' with score 3.757e-01
```

indicates that the condition was integrated into the system as a 'parse' and is available for further use after the GLS module has terminated. Not only these dependency analyses are retained, but all intermediate solutions that are produced. Finally, some statistics are printed of the performance profile of the GLS module. There are three lines of particular interest.

```
total          :    1700 ms  ...
solution       :     910 ms  ...  53.53 %  ...  3.757e-01 score
first solution :     780 ms  ...  45.88 %  ...  8.651e-06 score
```

The processing took 1700 milliseconds in total. The first intermediate solution, with no hard constraint violations, was found after 45.88% of the time. The final dependence analysis was determined after 53.53% of the time. The remaining time was used to fulfill the stated termination criteria.

b) Integration of agent heuristics

```
cdgp> gls -g1 3 -g2 0.03 -n 0.6 -dt 1000
```

The penalty weight adjustment parameter `-g1 3`, gives the control heuristics a relatively strong influence over the local gradients of the augmented cost function. The adjustment parameter `-g2 0.03` strengthens the variable allocations used in a local extreme point. (?) The parameter `-dt 1000` sets an upper limit to the number of iterations of the GLS heuristics. If this limit is reached without finding a better solution, the GLS agent returns the best state found.

c) Unary pruning in GLS

```
cdgp> gls -n 0.6 -up 2
```

The parameter `-up 2` means that the lowest 2% of each domain are deleted, if a local extreme point is found. Since GLS sorts the domains using the augmented cost function, more promising values are found in the upper range of a domain and more expensive values in the lower part. Unary pruning is a particularly useful heuristic for problems with very large search spaces.

d) Two co-operating GLS agents

```
cdgp> gls -g1 0.5 4 -g2 0.03
```

When two or more values are given to a parameter, a separate agent is started for each. In the example above, two agents are used: agent 0 with the cost function reduced by half (`-g1 0.5`), and agent 1 with the cost function increased by a factor four (`-g1 4`). One effect of this is that agent 0 needs a larger number of cycles to reach the same domain configuration as an agent 1, and the two agents will use completely different routes through the search space.

e) Two competing GLS agents

```
cdgp> gls -n 0.01 0.6 -comp
```

The parameter `-comp` prevents the distribution of data about the search results between the agents in the pool. It is unclear if this is useful or not.

f) Two sequential GLS agents

```
cdgp> gls -n 0.01 0.6 -from 0 9999 -t 1000 99000
```

It is possible to run agents sequentially, by giving them non-overlapping time limits. Here, agent 0 (`-n 0.01 -from 0 -t 1000`) runs alone, since agent 1 (`n 0.6 -from 9999 -t 99000`) begins running only at cycle 9999. If no agent is running, the next agent in line is started. Agent 1 thus begins his search prematurely, when agent 0 stops, although the cycle 9999 has not been reached.

1.2.15 The command 'help'

The parameters to the command '`help`' are names of other commands, whose help text is shown. If no parameters are given, help for all the commands is provided.

1.2.16 The command 'hierarchy'

Shows some or all hierarchy definitions that are loaded. If no parameters are given, all definitions are listed, otherwise only those specified.

```
cdgp> hierarchy ont
ont ->
  top(
    animate(
      human
      animal
    )
    inanimate(
      thing
      building
    )
  )
;
```

1.2.17 The command 'hook'

This command is not crucial for the normal use of `cdg`. It used to turn on and off certain functions in the system, mainly for presentation purposes.

The most important hook functions are `printf` and `flush`, which control the standard output. In the line oriented `cdg` this is the terminal from which `cdg` was started. In `Xcdg` it is a special window of the graphical interface.

The other hooks that are available at present are mostly for communication with the graphical interface.

Syntax: `hook [ on|off|reset | <HookName> [on|off|reset] ]`

All the hook functions (except `printf` and `flush` which are treated separately) can be switched on/off with `hook on|off`. Individual functions can be controlled by specifying their name as parameter. Their counters, which keep track of the number of calls, can also be reset.

```
cdgp> hook

  No Name                 State    Count
  ----------------------------------------
  00 buildnodes           disabled 0
  01 eval                 disabled 0
  02 netsearch            disabled 0
  03 printf               enabled  3672
  04 flush                enabled  100
  05 agenda               disabled 0

  hooking disabled.
```

1.2.18 The command 'incrementalcompletion'

This instruction starts an incremental search for dependency analyses of the word graph that is indicated as parameter. In each iteration the structure of the previous found sentence prefix is used as reference point for the further search.

The incremental search is configurable through the CDG variable `icparams`.

1.2.19 The command 'inputwordgraph'

This command is used to define linear word graphs on the command line. The entered sentences are automatically designated `wordgraph0`, `wordgraph1`, etc. and can be used immediately in the further processing.

```
cdgp> inputwordgraph die Katze jagt den Hund
INFO: wordgraph id: wordgraph0, #arcs: 5
cdgp>
```

1.2.20 The command 'isearch'

This command has been superseded by the command 'incrementalcompletion'.

1.2.21 The command 'level'

The command 'level' lists some or all loaded level definitions. If no parameters are given it lists all, otherwise only those specified.

```
cdgp> level
list of level declarations

  0 shown  used   SYN # VFIN SUBJ OBJA PMOD PN DET AMOD GMOD ADV JUNK;
  1 shown  used   SEM # ROOT AGENT THEME LOCATION DIRECTION PREP DUMMY;

number of binary constraints between values of given levels:

      |  0   1
   ---|---------
     0 | 28   6
     1 |  6   4

Level SYN uses 5 features: syn:gen syn:pers syn:num syn:cat syn:case
Level SEM uses 1 feature: sem:cat
```

For each level it is indicated whether it is used and/or shown (cf. the commands 'showlevel' and 'uselevel'), the label of the level, and the characteristics that defines it. Then comes a table indicating how many binary constraints there is between every pair of levels. Finally, the word attributes used are also listed, for each level.

1.2.22 The command 'levelsort'

This command defines the order of the levels in the grammar, used for sorting the constrain nodes. This is used to optimize the run time behaviour of the search. Sorting can be turned off with the command 'set sortnodes off'. If the command 'levelsort' are to be used to sort the nodes, then the command 'set sortnodes prio' should be given first. It is better to sort more *independent* and *important* variables (or constraint nodes) before purely technically motivated levels. Thus it makes more sense to find the syntactic assignment of all nodes, before instantiating the mirror planes of the syntax. Without sorting, all the levels are analysed for one word before moving on to the next word, so if there is a hard conflict on the syntax level, all the other levels for the earlier word were analysed unnecessarily. In other words sorting the levels permits earlier recognition of conflicts.

When the grammar files are loaded, the levels are initially sorted after the number of constraints per level, which gives an optimal sort order in many cases. In other cases, it is a poor choice with a negative impact on the search performance.

The command 'levelsort' takes as parameters a complete list of all level names, in the desired order. If given with no parameter, it lists the active sort order.

```
cdgp> levelsort
INFO: order of levels:  SYN DET VC1 VC2 DOM SEM
cdgp> levelsort SYN SEM DOM DET VC1 VC2
INFO: order of levels:  SYN SEM DOM DET VC1 VC2
cdgp>
```

1.2.23 The command 'lexicon'

The command 'lexicon' lists some or all of the loaded lexicon entries. If no parameters are given, all entries all listed; otherwise only those specified as parameters. Either lexicon names, or word forms can be specified. In the latter case all the entries for the word, if it is ambiguous, are listed.

```
cdgp> lexicon Katze
Katze_nomsg := Katze : [syn:[cat:noun, num:sg, case:nom, pers:3, gen:fem]];
Katze_accsg := Katze : [syn:[cat:noun, num:sg, case:acc, pers:3, gen:fem]];
Katze_datsg := Katze : [syn:[cat:noun, num:sg, case:dat, pers:3, gen:fem]];
Katze_gensg := Katze : [syn:[cat:noun, num:sg, case:gen, pers:3, gen:fem]];
```

1.2.24 The command 'license'

This command displays the software license of the system.

1.2.25 The command 'load'

The command 'load' loads one or more files with constraints, level definitions, lexicon entries, word graphs, analyses, hierarchy definitions and statistical parameters. The different types of data can be mixed arbitrarily. The respective number of inputs are shown during loading,

in the order given above. If any errors occur while loading, then *none* of the inputs are accepted. Warnings are only for information.

If the name of an input file ends in `.m4`, then the file contents are first processed by the pre-processor m4 and afterwards by `cdg`.

If a grammar structure is loaded that has the same type and designator as a structure already in memory, then the older version is over-written.

```
cdgp> load test/all
file 'test/all' loaded: 12/2/13/3/2/0/0
```

If a loaded file contains `#pragma` commands, they are executed only after the load has succeeded. This is so that an input file can specify both a grammar element and a command that references it.

1.2.26 The command 'ls'

The command 'ls' corresponds to the operating system command 'ls -laF'.

```
cdgp> ls /home/ingo/dawai/test/
total 118
drwxr-x---   2 ingo     nats         512 May 15 10:43 ./
drwxr-x---  13 ingo     nats         512 May 15 17:28 ../
-rw-r-----   1 ingo     nats        5048 May  7 09:46 all.cdg
-rw-r-----   1 ingo     nats        3316 May 13 14:13 berlin-sorten.cdg
-rw-r-----   1 ingo     nats        4245 May 12 09:57 berlin.cdg
-rw-r-----   1 ingo     nats        5039 May 13 09:33 dom.cdg
-rw-r-----   1 ingo     nats        4489 May  7 09:46 fail.cdg
-rw-r-----   1 ingo     nats        2431 May 15 10:43 frau.cdg
-rw-r-----   1 ingo     nats        2787 May  7 09:46 menzel.cdg
-rw-r-----   1 ingo     nats         989 May  7 09:46 mini-menzel.cdg
-rw-r-----   1 ingo     nats        2139 May  2 16:51 parkplatz.cdg
-rw-r-----   1 ingo     nats        4250 May  6 17:50 parkplatz2.cdg
```

1.2.27 The command 'net'

The commands 'net' activates one or more constraint nets. If no parameters are given, all nets are activated, otherwise, only those specified as parameters.

```
cdgp> net net0
------------------------------------------------------------
    id: net0
 state: 0
 nodes:
 0 die_sg_fem_nom(0,1)-SYN 0:
 1 die_sg_fem_acc(0,1)-SYN 2:
   DET-Frau_sg_acc(1,2)[1]
   DET-Tochter_sg_acc(4,5)[1]
 2 Frau_sg_nom(1,2)-SYN 1: SUBJ-sehen_tr_sg_3_pres(2,3)[1]
```

```
 3 Frau_sg_acc(1,2)-SYN 1: OBJ-sehen_tr_sg_3_pres(2,3)[1]
 4 sehen_tr_sg_3_pres(2,3)-SYN 1: ROOT-NIL[1]
 5 die_sg_fem_nom(3,4)-SYN 0:
 6 die_sg_fem_acc(3,4)-SYN 2:
   DET-Frau_sg_acc(1,2)[0.05]
   DET-Tochter_sg_acc(4,5)[1]
 7 Tochter_sg_nom(4,5)-SYN 1: SUBJ-sehen_tr_sg_3_pres(2,3)[1]
 8 Tochter_sg_acc(4,5)-SYN 1: OBJ-sehen_tr_sg_3_pres(2,3)[1]
#nodes: 7/9
#paths: 4
values: #min 1, #max 2, #total 9, average  1.29
#edges: 64
-------------------------------------------------------------
```

For those nodes in the net that are ambiguous, the number of lexemes represented by the node are given in braces (‘{}’):

```
[...]

10 krankenhaus_nom(1-2)/SYN 3:
SUBJ-->liegt_3_sg(2,3)[1]
SUBJ-->liegt_2_pl(2,3)[0.007]
PN-->in(3,4)[2.5e-06]
11 krankenhaus_acc(1-2)/SYN 3:
SUBJ-->liegt_3_sg(2,3)[0.05]
SUBJ-->liegt_2_pl(2,3)[0.00035]
PN-->in(3,4)[5e-05]
12 krankenhaus_dat(1-2)/SYN 3:
SUBJ-->liegt_3_sg(2,3)[0.05]
SUBJ-->liegt_2_pl(2,3)[0.00035]
PN-->in(3,4)[0.001]
⟹ 6 krankenhaus{3}(1-2)/SEM 4:  ⟸
AGENT-->liegt_3_sg(2,3)[0.0121]
THEME-->liegt_3_sg(2,3)[1]
AGENT-->liegt_2_pl(2,3)[0.0121]
THEME-->liegt_2_pl(2,3)[1]
21 liegt_3_sg(2-3)/SYN 1:  VFIN-NIL[1]
22 liegt_2_pl(2-3)/SYN 1:  VFIN-NIL[1]
⟹ 26 liegt{2}(2-3)/SEM 1:  ROOT-NIL[1] ⟸

[...]
```

1.2.28 The command ‘netdelete’

The command ‘netdelete’ deletes one or more constraint nets from the system. If no parameter is given, all nets are deleted, otherwise only those that are specified.

```
cdgp> netdelete net0
```

1.2.29 The command ‘netsearch’

The command ‘netsearch’ performs a search in the specified constraint net. For the time being, the following options are available.

- 'branchbound': a best-first search where partial solutions with a lower score than the best total result so far, are rejected (branch and bound). The two additional parameters are the maximum size of the agenda, and an absolute threshold for evaluations. Note: If this search method is used with constrains that have a cost larger than one, the result is undefined.

  This is the standard search method.

- 'fullsearch': This search method is similar to 'branchbound', except that no partial solutions are eliminated.

```
cdgp> netsearch net0 branchbound 1000
INFO: agenda size set to 1000
INFO: solution with score 1.000e+00 found
INFO: the search took 53 steps

INFO: net: net0, wordgraph: KORR/1
INFO: agenda size: 33/1000, 1 solution(s) with score 1.000e+00:
------------------------------------------------
   +----- compared to annotation `|' label, `-' dependency
   |
   | +--- modifier `-' lexical entry, `*' word form
   | |+-- label
   | ||+- modifiee `-' lexical entry, `*' word form
   | |||
00       #5 der_mas_sg_nom(0-1)/SYN--DET-->parkplatz_nom(1-2) (1.000e+00)
01       #20 parkplatz_nom(1-2)/SYN--SUBJ-->liegt_3_sg(2-3) (1.000e+00)
02       #35 liegt_3_sg(2-3)/SYN--VFIN-->NIL (1.000e+00)
03       #40 hinter(3-4)/SYN--PMOD-->liegt_3_sg(2-3) (1.000e+00)
04       #61 der_fem_sg_dat(4-5)/SYN--DET-->fleischerei(5-6) (1.000e+00)
05       #68 fleischerei(5-6)/SYN--PN-->hinter(3-4) (1.000e+00)

INFO: #unary best: 6/6 6 0 0 0 0 0 0 0 0 0
------------------------------------------------
INFO: net: id net0, wordgraph KORR/1
      #nodes 15, #edges 210
      #evaluations: 0 unary, 0 statistics, 0 binary
      #values: min 1, max 7, total 70, average  4.67
      cache: size 70, #hits 116, 1.7 per each
```

When the search has finished, the best scoring solutions are listed. For each of them, all connections are listed, and for each connection is listed its sequence number, the arc data, and the unary score. The arc data consists of the label and position of the sub-ordinated lexeme, the level name, the arc label, and the label and position of the super-ordinated lexeme (or `NIL` if the arc points to a root).

After the list of connections it is shown how many of them had the highest unary score (in the example, all of them), and after that comes information about the underlying constraint network.

If the solution violates any constraints, then these are enumerated after the list of connections. The output

```
 syn_det_adj_numerus(1.000e-01): 00-01
```

means that the bindings 00 and 01 violates the binary constraint `syn_det_adj_numerus` with a value of 0.1.

A constraint can only be violated once by any particular set of edges. Given the following basic constraint

```
// A verb can not have two complements.
X:SYN, Y:SYN : verb_arity_two(?) : syn_verb : 0.95 :
  X^syn:cat=verb
  & X^id=Y^id
  & distance(X@id,X^id) > 0
  &  distance(Y@id,X^id) > 0
  -> X@id=Y@id;
```

and the sentence 'Ich dich sehe,' there is only one violation of the constraint, and the total score is only decreased once, even though both pairs of edges, $(0, 1)$ and $(1, 0)$, violates the constraint.

```
INFO: net: net1, wordgraph: wordgraph0
INFO: agenda size: 4/1000, 1 solution(s) with score 4.750e-01:
-------------------------------------------------
   +----- compared to annotation '|' label, '-' dependency
   |
   | +--- modifier '-' lexical entry, '*' word form
   | |+-- label
   | ||+- modifiee '-' lexical entry, '*' word form
   | |||
00       #0 ich(0-1)/SYN--SUBJ-->sehe(2-3) (1.000e+00)
01       #4 dich(1-2)/SYN--OBJA-->sehe(2-3) (5.000e-01)
02       #6 sehe(2-3)/SYN--VFIN-->NIL (1.000e+00)

INFO: #unary best: 3/3 3 0 0 0 0 0 0 0 0 0

        syn_obja_pos(5.000e-01): 01
   verbzweitstellung(9.500e-01): 00-01
-------------------------------------------------
```

If several equivalent solutions were found, then the differences between successive solutions, or between solution and annotation, is shown in the columns in front of the connections.

If the command line option '-m' is given to the program, then those connections in the constraint network the were not selected are deleted after the search has finished.

1.2.30 The command 'newnet'

The command 'newnet' creates a new constraint net from a word graph, whose label must be given as a parameter. Information is displayed about lexeme-graphs, the label of the new net, the number of nodes and edges, and the scores of unary, statistic and binary constraints, as well as the domains of the nodes.

```
cdgp> newnet test1
INFO: lexem graph: #nodes 9, min 0, max 5, #paths 16
INFO: net: id net0, #nodes 9, #edges 64
      #evaluations 869/0/224
      values: #min 0, #max 2, #total 9, average  1.00
      cache: size 86, #hits 1728, 20.1 per each
```

For efficiency, the computed scored for the constraints are held in a cache. The final line in the output gives statistic data about the use of this cache.

1.2.31 The command 'nonspeccompatible'

This commands identifies constraints that might show undefined behavior if a dependency analysis with under-specified nodes is evaluated. At present such structures are created by the command 'isearch' only.

1.2.32 The command 'parsedelete'

Deletes parses. If the first parameter is `-w`, and the second is the name of a word graph, then all parse structures for that word graph are deleted. Structures derived from annotations are not deleted unless a third parameter `-f` is also given.

1.2.33 The command 'parses2prolog'

Writes all currently existing parses to the specified file in Prolog format as per the following (pseudo-)BNF. The number of annotations that were written to the file are printed on the screen.

```
<File>        ::= { <Parse> }*

<Parse> ::= parse(<Id>,<LaId>,<Words>,<Levels>,<Labels>,<Nvs>,<VSs>,<Nvl>,<VLs>).

<Id>,<LaId>   ::= <Quoted Prolog atoms>

<Words>       ::= <Prolog list of Word>

<Word>        ::= word( <From> , <To> , <String> , <Description> )

<From>,<To>   ::= <Prolog integers>

<String>,<Description> ::= <Quoted Prolog atoms>

<Levels>,<Labels> ::= <Prolog list of quoted atoms>

<VSs>         ::= <Prolog list of integers>

<VLs>         ::= <Prolog list of quoted atoms>

<Nvs>,<Nvl>   ::= <Prolog integers>
```

The lattice label (`LaId`) identifies the word graph, as in the output from `annos2prolog` (section 1.2.3). The two integers `<Nvs>` and `<Nvl>` gives the lengths of the lists `<VSs>` and `<VLs>`. These two numbers will always be the number of words (`Words`) times the number of levels (`Levels`).

1.2.34 The command 'printparse'

This command displays the specified dependency analysis in textual form.

1.2.35 The command 'printparses'

This command displays all the analyses of the specified constraint net, in textual form.

1.2.36 The command 'quit'

The command 'quit' terminates the program.

```
cdgp> quit
bye
%
```

1.2.37 The command 'shift-reduce'

This command starts a shift-reduce parser that tries to construct a valid structure on the defined main level (auxiliary levels are computed in isolation as needed).

The parameters are `key=value` pairs with the following meaning:

Key	Meaning	Default value
`beam <num>`	agenda size of local search	`1`
`net <name>`	constraint net to use	""
`policy <name>`	arbitration policy	`table`
`corpus <num>`	weight of corpus information	0.5

Defined values for `policy` are:

- `exact:RLslrS`: The exact action at the corresponding point in the program is used (where r=REDUCE, R=RIGHT, s=shift, L=LEFT). Implies `beam_width` == 1.
- `best`: prefer the transitions that result in the best Badness.
- `nivre`: prefer transitions in the order LEFT–RIGHT–REDUCE–SHIFT.
- `table`: the transition is chosen that occurred most often in similar parse states in the reference corpus.
- `hybrid`: the judgements of `best` and `table` are combined in non-obvious ways.

Shift-reduce parsing is highly experimental, but not under development; therefore there is little point in using it.

1.2.38 The command 'renewnet'

This command resets a constraint net to the original condition. All computed solutions are removed, all deleted elements are restored, and all parameters are given the original values. (This is sometimes useful after applying procedures that change the internal state of a constraint net, such as 'frobbing', or if too many values were pruned away.)

```
cdgp> newnet test1
INFO: lexem graph: #nodes 9, min 0, max 5, #paths 16
INFO: net: id net0, #nodes 9, #edges 64
      #evaluations 869/0/224
      values: #min 0, #max 2, #total 9, average  1.00
      cache: size 86, #hits 1728, 20.1 per each
cdgp> pruning net0
cdgp> renewnet net0
```

1.2.39 The command 'reset'

This command returns the system to the starting conditions. All grammar definitions and all computed analyses are removed from memory.

1.2.40 The command 'section'

This command gives an overview of the defined constraint classes. See also the commands 'activate' and 'deactivate'.

```
cdgp> section
active  ,     0 constraint(s), default
active  ,     5 constraint(s), syn
active  ,     5 constraint(s), sem
active  ,     2 constraint(s), map
```

1.2.41 The command 'set'

This command allows control over CDG variables. The following variables are available:

- 'CC'
- 'CFLAGS'
- 'INCLUDES'
- 'LD'
- 'LDFLAGS'
- 'LDLIBS'

  These variables correspond to shell variables used in compiling CDG constraints to C code.

- 'acoustics': Setting this variable currently has no effect.
- 'anno-categories': This is a comma-separated list of features that are considered important by the grammar writer. They influence the behaviour of CDG in two places:
  1. when renaming homonyms in the lexicon, the values of these features are used preferably to construct unique names for each entries. This means that the command `set anno-categories case,number,gender` ensures that your determiners will be renamed to `der_nom` and `der_dat` and not after some other feature.
  2. when creating annotations, the values of these features are always recorded for each word along with the word form, unless they are absent from the word.
- 'annodirs': This is the directory where annotation files are stored. If you put the annotation 'foobar' into the file 'foobar.cda' and point this variable to this directory, it will be autoloaded, e.g. when you issue the command `anno2parse foobar`.

  Files whose names end in this string are also detected, so you can have 'automatic-foobar.cda' and 'gold-foobar.cda' side by side, and they will both be loaded when appropriate.

  To avoid creating directories with millions of entries, annotations may be kept in numbered subdirectories instead of the directory itself, in batches of 10,000. If the name of an annotation contains a decimal number, e.g. 528754, then the subdirectory '052' of 'annodirs' will also be searched. (To deal with more than 10,000,000 sentences, please extend the program.)

  Autoloading of annotations works only when a particular annotation is explicitly named; the command `annotation s1` will autoload 's1.cda' if necessary, but mere `annotation` will only display all annotations presently in RAM; it will **not** autoload all available annotations.
- 'autocompare': If set, all intermediate solutions of 'netsearch' will immediately be evaluated for their recall.
- 'cache': Controls whether, during repeated evaluation of the same constraints, the computation is actually repeated, or the result are computed once and then stored in a cache. Use of the cache is normally switched on.
- 'capitalizable-categories': This should be a comma-separated list of lexical categories. The items should be possible values for the feature given by 'taggerCategoryPath'. If set, words of these categories can be retrieved from the lexicon in their normal form even when they appear capitalized in the input. For German, this often occurs with `ADJA` words (and sometimes `ART`).
- 'chunker': If set, the chunk parser defined by 'chunkerCommand' will be applied to every lexeme graph created. Its results will be accessible through the function 'chunk_head' and its associated features in the constraint language.
- 'chunkerCommand': The shell command to call the desired chunk parser.
- 'chunkerMode': If set to `real`, the chunker as defined by 'chunkerCommand' will be called on every lexeme graph. If set to `fake`, chunk information will be inserted by reading the corresponding annotation, i.e., faked. If set to `eval`, the real chunker will be called and evaluated against the annotation when the 'chunk' command is used.

- 'compound-categories': This should be a comma-separated list of lexical categories. The items should be possible values for the feature given by 'taggerCategoryPath'. If set, even compounds that are formed unmarked (without a hyphen) can be looked up in the lexicon by finding the simplex form if it belongs to one of these categories. For example, 'Großwesir' will be handled by looking up 'Wesir' and copying its features.

  See 'deduceCompounds' for detecting marked compounds.

- 'debug': Controls whether or not messages that have the tag `DEBUG` are shown. `DEBUG` messages are more detailed than `INFO` and give internal information that the normal user is not supposed to need for operating the program. Valid values are `on` and `off`. Default is `off`.

- 'deduceCompounds': If set, lexicon lookup is allowed to substitute base forms for marked compounds. This means that the query 'lexicon queen-empress' will succeed even if 'queen-empress' is not in the lexicon, by looking up 'empress' and assuming that the longer word carries the same features. The word 'actor/producer' is likewise deduced from the word 'producer'.

  This only works for compounds with a hyphen or a slash in them; see 'compound-categories' for detecting unmarked compounds.

- 'edges': This variable can be set to `on`, `off`, `all`, or `few`. It controls the production of edges between the nodes of a constraint network. The complete investigation of a constraint network is possible when this variable is set to `all`. **Note:** currently no solution methods use constraint edges, therefore there is no reason ever to set this variable.

- 'encode-umlauts': If this Boolean variable is set, text typed by the user on the command line is subjected to the following substitutions:

"a	⇒	ä
"o	⇒	ö
"u	⇒	ü
"A	⇒	Ä
"O	⇒	Ö
"U	⇒	Ü
"s	⇒	ß

  This allows you to type proper German umlauts even if your terminal doesn't pass eight-bit characters.

- 'error': Controls whether messages that have the tag `ERROR` are displayed or not. Valid values are `on` and `off`. The default is `on`. `ERROR` messages should not be suppressed since they indicate if an action is aborted.

- 'eval': This variable is not used.

- 'evalmethod': If set to `interpreted` (the default), constraints are evaluated by walking the internal representation of their logical formulas. If set to `compiled`, the compiled machine code is executed instead. **Note:** The constraint compiler is largely inoperable, and where functioning does not actually speed up evaluation, therefore it makes no sense to set this variable.

- 'featureHierarchy': This variable should be set to the name of a defined hierarchy of values. When comparing the analysis of an utterance to an annotation, this hierarchy is used to check any partially specified values in the analysis.

  Say that an annotation specifies an attribute of `gender=masc` for an adjective, but the lexicon only contains entries with the features `fem` and `not_fem` to save space, because the `masc` and `neut` forms are identical, and the parser chose the `not_fem` reading. Normally the comparison would count this as a mismatch. But if you point `featureHierarchy` to the name of a hierarchy which makes this relation explicit, like this,

  ```
  Features ->
          Gender -> fem neut masc,
          not_fem -> neut masc;
  ```

  then the verification engine allows `not_fem` as a less specified form of `masc`.

- 'hint': This variable is not used.

- 'icparams': Specifies the behavior of the command 'incrementalcompletion' in more detail. The value must be a string which can contain different flags and options.

Option	Meaning
`-l`	Set the main analysis level
`-f`	Set the name of the log file
`-a`	Set the agenda size for the search phase
`-1` – `-5`	Set individual heuristics in different classes

  This variable can for example be set as follows:

  ```
  set icparams '-l SYN -a 10000 -1 1 -2 1'
  ```

- 'ignorethreshold': 'frobbing' does not try to repair conflicts whose score exceeds this value. Therefore it should be set to a value between that of your error constraints and your diagnostic constraints.

- 'info': Controls whether messages with the tag `INFO` are shown or not. Valid values are `on` and `off`. The default is `on`.

- 'locale': The value of this variable is used as the `LC_CTYPE` for string operations that `cdg` performs. If, for instance, you parse raw German text, you will need to set this to `de_DE` or something similar so that `cdg` will know that 'Über' at the beginning of a sentence may be an instance of 'über'.

- 'normalization': If set to `off`, partial results in the search are only compared on the basis their values. If set to `linear`, `square`, `depth`, or `breadth`, longer partial solutions are given a higher score. Only affects the 'netsearch' command.

- 'peekvaluemethod': If set to `compiled` (the default), the features of a lexicon item are pre-computed into a lookup table. If set to `interpreted`, they are found by descending into the structure of the lexicon item in memory, which takes somewhat longer. **Note:** although the possible values are the same, this has nothing whatsoever to do with the 'evalmethod' variable. In particular, pre-computed feature lookup can be used with compiled as well as with interpreted constraints. It is fully functional and gives a small but consistent speedup at no price. Therefore there is no reason ever to set this variable.

- 'preprocessor': This variable contains the absolute path names of the pre-processor used by the command 'load'. Only files with the suffix 'prepsuffix' are pre-processed. Default values are `/opt/bin/m4` under Solaris, and `/usr/bin/m4` under Linux. Since the options `-p` and `-S` are given to this program, other pre-processors are not suitable.
- 'prepsuffix': Only files that end with this suffix are processed by the 'preprocessor'.
- 'profile': Controls whether or not additional information about the time requirements of individual actions are reported. The measurement accuracy is only 10ms. Possible values for this variable are `on` and `off`. Default is `off`.
- 'progress': Controls whether messages that have the tag `PROGRESS` are shown. The default is `off`.
- 'prolog': This variable has no effect.
- 'searchmodifiesnet': Controls whether searching a net may change it or not. Valid values are `on` and `off`. Default is `off`.
- 'searchresult': Controls whether results are printed immediately after completing the search of a constraint network. The default value is `on`. See also the command `printsolutions`.
- 'shift-reduce-table': This is the name of the file used by 'shift-reduce' with the policy `table`.
- 'showdeleted': This variable corresponds to the command line option '-d' and controls whether the display of a constraint net (see the command 'net') includes deleted values, or not. If they are included, they are surrounded by square brackets.
- 'sortnodes': Controls whether or not the nodes are sorted when the command 'newnet' constructs a new net. Possible values are `off`, `prio`, and `smallest`. With `prio` the order given by the command 'levelsort' is used. The value `smallest` means that the nodes are sorted according to increasing domain size, i.e. the nodes with fewer possibilities comes first. Default is `off`.
- 'statistics': Corresponds to the command line option '-s' and controls the use of statistical parameters. Should not be used.
- 'subsumesWarnings': Controls the printing of warnings, if the function of 'subsumes' causes a type error. Can be set to `full` or `sloppy`.
- 'templates': Controls the use of lexicon templates. If set to 'never', templates are never used even if they are defined. If set to 'always', all matching templates will always auto-generate lexicon items. If set to 'ifneeded' (the default), a matching template will be used if there is no ordinary lexicon present for a form; but if there is even one normal item, templates will not be used even if they would match. If set to 'bycategory', a matching template will be used if there is no ordinary item *of the same category*. This only works if 'taggerCategoryPath' is set, otherwise the behaviour reverts to 'ifneeded'.
- 'taggerCategoryPath': The name of the feature that specifies the part of speech for lexicon items. For instance, if your lexicon items look like this

```
ich := [ syn:[cat:pron, num:sg, pers:1, case:nom ],
         sem:[cat:ICH] ];
```

then 'taggerCategoryPath' should be set to `syn:cat`. POS tagging only works when this variable is set.

- 'taggerCommand': The shell command to call the desired POS tagger. It must be a filter (read from STDIN and print to STDOUT).
- 'taggerIgnoreImpossible': If set, the POS scores assigned to categories that are not actually possible according to your lexicon are ignored. (This will issue warnings saying 'Your lexicon does not allow 'furl' to be a verb', but otherwise it has no effect unless 'taggerNormalizeToOne' is also set.)
- 'taggerNormalizeToOne': If set, POS scores are normalized so that the highest score assigned becomes 1. (Usually taggers emit true probabilities, so that even the preferred value has a score lower than 1.) This is the recommended setting.
- 'timelimit': Indicates the number of milliseconds after which the search for a dependency analysis is aborted. The value zero means unlimited.
- 'unaryFraction': This variable can take a value between 0 and 1. Normally it has the value 1. If it is set to a smaller value, then a limited pruning is done already when the constraint network is constructed. In each constraint node, all values that fall below a certain threshold are deleted. This threshold is the product of (1−`unaryFraction`) and the best value in the node.
- 'usenonspec': Controls whether dependency edges that are explicitly underspecified w.r.t. their regent are built into constraint nets or not.
- 'verbosity': Corresponds to the command line option '-q' and controls whether additional information is printed. (Its value is actually the bitwise superimposition of the internal representation of those output flags that are currently set, e.g. 'INFO', 'WARNING' and 'ERROR', but you should set these flags under their symbolic name instead.)
- 'warning': Controls whether messages that have the tag `WARNING` are printed, or not. Possible values are `on` and `off`. The default is `on`. The difference between `WARNING`s and `ERROR`s is that execution is aborted when the latter, but not the former, occur.
- 'xml': Controls the printing of structured output in XML format. This output goes to log files, not the screen. The default value is `off`.

```
cdgp> set verbosity on
```

1.2.42 The command 'showlevel'

Display of the levels that are given as parameters to this command is toggled on or off. When the display of a level is 'off' it is not included when information about the analysis is printed. This command has no effect on the internal use of the level in computations.

See also the command 'uselevel'.

```
cdgp> showlevel SYN
INFO: level 'SYN' isn't shown now
```

1.2.43 The command 'status'

The command 'status' prints information about the system status.

```
cdgp> status
         constraints: 10
  level declarations: 1
     lexical entries: 7
          wordgraphs: 1
         annotations: 0
         hierarchies: 0
          parameters: 0

     constraint nets: 0

           verbosity: 1319
 show deleted values: no
      use statistics: no
 search modifies net: no
    normalize scores: no
   subsumes warnings: full
         build edges: yes
unary pruning factor: 1.000000
             caching: yes
          sort nodes: no

        preprocessor: /opt/bin/m4
    1 loaded file(s):
                       /home/ingo/dawai/test/frau.cdg
```

1.2.44 The command 'tagger'

With the argument `on`, this command switches the POS tagger on, and with the argument `off` it switches the tagger off. The variable 'taggerCommand' must have a suitable value.

1.2.45 The command 'testing'

This command executes code that is, by definition, temporary and of no interest whatsoever to the user.

1.2.46 The command 'useconstraint'

This command toggles the use of the specified constraints on or off. The effect of setting the use of a constraint 'off' is as if had never been defined.

```
cdgp> useconstraint syn_circle'
INFO: constraint 'syn_circle' isn't used now
```

1.2.47 The command 'uselevel'

This command toggled the use of the specified levels on or off. The effect of setting the use of a level 'off' is as if had never been defined.

See also the command 'showlevel'.

```
cdgp> uselevel SYN
INFO: level 'SYN' isn't used now
```

1.2.48 The command 'uselexicon'

This command directs `cdg` to an external database of lexicon items. The parameter must be a string `foo`, and the file of CDG input and the index into this file (created with the bundled `indexer` tool) will then be searched as `foo.cdg` and `foo.db`.

Giving this command multiple times is not cumulative; only the input file given last is searched. See 'closedb' on closing the external data base.

1.2.49 The command 'verify'

Compares the parse specified, or the last parse created if there is no parameter, to the annotation with the same label. Four correctness measures are used:

- how many dependency edges have been established perfectly?
- how many dependency edges have been established perfectly except for the lexical reading of the regent?
- how many dependency edges have been established perfectly except for lexical readings?
- how many dependency edges have been established perfectly except for lexical readings and labels?

1.2.50 The command 'version'

The command 'version' prints the version number of the program.

```
cdgp> version
CDG parser v0.10beta (Build 8)
Ingo Schröder ingo.schroeder@informatik.uni-hamburg.de
Type 'help' for help.
```

1.2.51 The command 'weight'

With no argument, displays all constraint weights. With one argument, displays the weight of the specified constraint. With two arguments, sets the weight of the specified constraint to the second argument. With three arguments, the first must be the name of a constraint, the second the string `absolute` and the third a penalty. The specified constraint will have its penalty changed to this constant penalty, even if it was variable before.

```
cdgp> weight cycle
cycle: 0.000e+00
cdgp> weight cycle 1
cdgp> weight cycle
cycle: 1.000e+00
cdgp>
```

1.2.52 The command 'wordgraph'

The command 'wordgraph' lists some or all loaded word graphs. If no parameters are given, all word graphs are listed. Otherwise, only those whose label is given as parameter, or that contains a parameter as the word.

```
cdgp> wordgraph worterkenner_M/1/1
worterkenner_M/1/1 : M/1/1 :
 0  1          pferde 0.000000
 0  2              er 0.000000
 1  3         fressen 0.000000
 2  3       befestigt 0.000000
 3  4            gras 0.000000

cdgp> wordgraph dem
KORR/2 : KORR/2 :
   0  1             die 0.000000
   1  2     fleischerei 0.000000
   2  3             ist 0.000000
   3  4             vor 0.000000
   4  5             dem 0.000000
   5  6       parkplatz 0.000000

KORR/14 : KORR/14 :
   0  1             vor 0.000000
   1  2             dem 0.000000
   2  3       parkplatz 0.000000
   3  4           liegt 0.000000
   4  5             die 0.000000
   5  6     fleischerei 0.000000

SYN/vk : SYN/vk :
   0  1             die 0.000000
   1  2         kirchen 0.000000
   2  3           liegt 0.000000
   3  4           neben 0.000000
   4  5             dem 0.000000
   5  6       parkplatz 0.000000
```

1.2.53 The command 'writeannotation'

Finds the best analysis of the named constraint net and writes it to disk in a format suitable for later CDG input.

1.2.54 The command 'writenet'

This command produces a LaTeXrepresentation of the specified constraint network, and stores it in the specified file.

1.2.55 The command 'writeparses'

Produces a LaTeXrepresentations of all dependency analyses of the specified constraint networks, and stores them in the specified file.

1.2.56 The command 'writewordgraph'

Produces a LaTeXrepresentation of the specified word graphs, and stores them in the specified file.

1.3 Example session

Here is shown an example of how an utterance can be analysed with the 'cdg' system.

```
% cdg

CDG parser
Ingo Schroeder ingo.schroeder@informatik.uni-hamburg.de
Type 'help' for help.

cdgp> load test/menzel
file 'menzel.cdg' loaded: 12/2/8/11/0/0/0
cdgp> status
         constraints: 12
  level declarations: 2
     lexical entries: 8
          wordgraphs: 11
         annotations: 0
         hierarchies: 0
          parameters: 0

     constraint nets: 0

           verbosity: 1319
 show deleted values: no
      use statistics: no
 search modifies net: no
```

```
    normalize scores: no
   subsumes warnings: full
         build edges: yes
unary pruning factor: 1.000000
             caching: yes
          sort nodes: no

        preprocessor: /opt/bin/m4

    1 loaded file(s):
                       menzel.cdg
```

First, a file is loaded, that contains constraints, level definitions, lexicon entries, and word graphs from Menzel (1995).

```
cdgp> wordgraph M/1/1
M/1/1 : M/1/1 :
 0  1          pferde 0.000000
 1  2         fressen 0.000000
 2  3            gras 0.000000
cdgp> newnet M/1/1
INFO: lexem graph: #nodes 3, min 0, max 3, #paths 1
INFO: net: id net0, #nodes 6, #edges 18
      #evaluations 230/0/188
      values: #min 1, #max 2, #total 10, average  1.67
cdgp> net net0
------------------------------------------------------------
    id: net0
 state: 0
 nodes:
 0 pferde(0,1)-SYN 2: SUBJ-fressen(1,2)[1] OBJ-fressen(1,2)[1]
 1 pferde(0,1)-SEM 2: AG-fressen(1,2)[1] PAT-fressen(1,2)[0.7]
 2 fressen(1,2)-SYN 1: ROOT-NIL[1]
 3 fressen(1,2)-SEM 1: ROOT-NIL[1]
 4 gras(2,3)-SYN 2: SUBJ-fressen(1,2)[0.03] OBJ-fressen(1,2)[1]
 5 gras(2,3)-SEM 2: AG-fressen(1,2)[0.1] PAT-fressen(1,2)[1]
#nodes: 6/6
#paths: 1
values: #min 1, #max 2, #total 10, average  1.67
#edges: 18
------------------------------------------------------------
```

Then, a constraint net is created, and its label (`net0`) returned. The six constraint nodes can produce a solution, since they were not all eliminated by violating unary constraints with a score of 0.0. The constraint network has 18 edges: from each node there is one edge to a node with the same lexeme and a different level, and two edges to nodes with different lexemes and the same level. There were 230 evaluations of unary constraints, and 188 of binary ones; no statistic evaluations were made. With six nodes and ten values, there are between one and two values per node with an average of 1.67.

```
cdgp> netsearch
INFO: using most recently created net ‘net0’
```

```
INFO: solution with better score 1.000e+00 found

INFO: agenda size: 5/1000, 1 solution(s) with score 1.000e+00:
-------------------------------------------------
   +--- modifier '-' lexical entry, '*' word form
   |+-- label
   ||+- modifiee '-' lexical entry, '*' word form
   |||
00     pferde/SYN(0-1)->SUBJ->fressen(1-2)
01     pferde/SEM(0-1)->AG->fressen(1-2)
02     fressen/SYN(1-2)->ROOT->nil
03     fressen/SEM(1-2)->ROOT->nil
04     gras/SYN(2-3)->OBJ->fressen(1-2)
05     gras/SEM(2-3)->PAT->fressen(1-2)

INFO: #violated constraints: 0/0
-------------------------------------------------
```

The command 'netsearch' starts a global search for the best solution. A solution is found, above, with a score of 1.0.

The command 'netsearch' is the safest way of testing a constraint grammar. The 'classical' method of making the nodes and edges consistent also works, however, as can be seen from the following.

```
cdgp> newnet M/1/1
INFO: lexem graph: #nodes 3, min 0, max 3, #paths 1
INFO: net: id net1, #nodes 6, #edges 18
      #evaluations 230/0/188
      values: #min 1, #max 2, #total 10, average  1.67
cdgp> nodeconsistency net1 absolute 0.2
INFO: limit value set to 0.200000
cdgp> net net1
------------------------------------------------------------
    id: net1
 state: 0
 nodes:
 0 pferde(0,1)-SYN 2: SUBJ-fressen(1,2)[1] OBJ-fressen(1,2)[1]
 1 pferde(0,1)-SEM 2: AG-fressen(1,2)[1] PAT-fressen(1,2)[0.7]
 2 fressen(1,2)-SYN 1: ROOT-NIL[1]
 3 fressen(1,2)-SEM 1: ROOT-NIL[1]
 4 gras(2,3)-SYN 2: [SUBJ-fressen(1,2)[0.03]] OBJ-fressen(1,2)[1]
 5 gras(2,3)-SEM 2: [AG-fressen(1,2)[0.1]] PAT-fressen(1,2)[1]
#nodes: 6/6
#paths: 1
values: #min 1, #max 2, #total 8, average  1.33
#edges: 18
------------------------------------------------------------
```

In making the network node-consistent, two possible solutions were eliminated. The nodes with the indexes 4 and 5 now have only one possible value each.

```
cdgp> arcconsistency net1 rowcolumn 0.4
```

```
cdgp> net net1
------------------------------------------------------------
    id: net1
 state: 6
 nodes:
 0 pferde(0,1)-SYN 2: SUBJ-fressen(1,2)[1] [OBJ-fressen(1,2)[1]]
 1 pferde(0,1)-SEM 2: AG-fressen(1,2)[1] [PAT-fressen(1,2)[0.7]]
 2 fressen(1,2)-SYN 1: ROOT-NIL[1]
 3 fressen(1,2)-SEM 1: ROOT-NIL[1]
 4 gras(2,3)-SYN 2: [SUBJ-fressen(1,2)[0.03]] OBJ-fressen(1,2)[1]
 5 gras(2,3)-SEM 2: [AG-fressen(1,2)[0.1]] PAT-fressen(1,2)[1]
#nodes: 6/6
#paths: 1
values: #min 1, #max 1, #total 6, average  1.00
#edges: 18
------------------------------------------------------------
```

As expected, the remaining ambiguity is removed by transferring the net into an edge-consistent condition. The one remaining solution is the desired result.

1.4 Grammar elements

A constraint grammar can consist of lexical entries, level declarations and constraints, hierarchy definitions, and data maps.[3] Unless at least one level of analysis is declared, no parsing is possible. The utterances to be analysed come in the form of word graphs and annotations with category symbols and dependencies.

In addition to grammar elements, input files may contain cdg commands to be executed at load time. Any line beginning with '`#pragma` ' is interpreted as a cdg command. For instance, a grammar of German would say

```
#pragma set locale de_DE
```

to ensure that it is always run with the correct locale set.

The Tables 1.1 to 1.8 specify the syntax of the inputs. Block comments begin with '`/*`,' end with '`*/`,' and can be nested. Line comments begin with '`%`' or '`//`' and continue to the end of the line. Symbols are either sequences of alphabetical characters, numbers, underscore, and characters with the eight bit set, or sequences of arbitrary characters enclosed in single (') or double (") quotation marks.[4]

Quoted strings can span more than one line if the end of each line is marked with a backslash (\).

[3] There are also statistical parameters which should not be used.

[4] In earlier versions only seven bit characters were permitted. Since umlauts could not then be written, the double quotation mark (") was also allowed. Strings were thus sequences of the characters `[a-zA-Z0-9_"]`. This prevented the use of " as a string delimiter, so the character ' could not be allowed in a string. In the current version all printable characters can occur in strings. The old behavior can be activated by changing the specification of `libcdg` in the rule for producing `scanner.l` in the Makefile, according to the comment.

NUMBER	::=	`[0-9]+`
	|	`[0-9]*.[0-9]+`
STRING	::=	`'[^\n']*'`
	|	`"[^\n"]*"`
	|	`[_a-zA-Z\x80-\xff][a-zA-Z_0-9\x80-\xff]*`

Table 1.1: Lexical units (defined by regular expressions)

1.4.1 Levels of analysis

Level declarations are written like this:

```
SYN # ROOT, SUBJ, OBJ;
SEM # ROOT, AG, PAT;
```

A level declaration can be have *properties* with extra information:

```
Pragmatik [schwierig=ja, wichtig=ja] # label1, label2, label3;
```

These are pairs of strings, and are currently used in only one way by the software: if a level carries the property 'mainlevel', then that level is displayed by `xcdg` as a tree with all other edges as additional arcs.

1.4.2 Constraints

Here are some example constraints:

```
// Kongruenz Subjekt - Verb
{X} : sy2 : syn : 0.1 :
  X.level=SYN & X.label=SUBJ -> X@num=X^num;

// Korrespondenz Subj. - Ag. und Obj. - Pat.
{X:SYN, Y:SEM} : ss1 : map : 0.2 :
  X^id=Y^id & X@id=Y@id ->
    (X.label=SUBJ & Y.label=AG) | (~X.label=SUBJ & ~Y.label=AG);
```

INPUTSEQ	::=	*empty*
	|	INPUTSEQ LEXICALENTRY ';'
	|	INPUTSEQ LEVELDECL ';'
	|	INPUTSEQ CONSTRAINT ';'
	|	INPUTSEQ WORDGRAPH ';'
	|	INPUTSEQ ANNOENTRY ';'
	|	INPUTSEQ HIERARCHY ';'
	|	INPUTSEQ PARAMETER ';'

Table 1.2: Input syntax (EBNF)

Label	::=	String
LabelList	::=	LabelList ‘,’ Label
	|	Label
LevelDecl	::=	LevelId ‘[’ PropertyList ‘]’ ‘#’ LabelList
	|	LevelId ‘#’ LabelList
LevelId	::=	String
Property	::=	String ‘=’ String
PropertyList	::=	PropertyList ‘,’ Property
	|	Property

Table 1.3: Level declarations (EBNF)

A constraint begins with the declaration of the variables that it concerns. Unary and binary constraints are allowed, but no constraints of higher arity. The name of a constraint variable is arbitrary, but `X` and `Y` are recommended. In binary constraints, the two variables are separated with a comma (or something else, see below).

Instead of just the variable name, a constraint can be confined to variables of a particular level by adding `:<levelname>` to the variable name. The effect of `{X:SYN}` rather then just `{X}` is exactly as if the constraint body were prefixed with `X.level=SYN ->`, but the former way is more efficient to evaluate.

A variable declaration can be confined further by replacing the colon with another connector (e.g. `{X!<levelname>}`) that symbolizes the direction of an edge. If a variable declaration uses such a direction indicator, then the constraint is applied only to those edges that fulfill

Attr	::=	String
	|	Number
AttrValue	::=	Attr ‘:’ Value
Conjunction	::=	Conjunction ‘,’ AttrValue
	|	AttrValue
Disjunction	::=	Disjunction ‘—’ Value
	|	Value
LexicalEntry	::=	Word ‘:=’ Value
LexicalEntry	::=	Word ‘= ’ Value
Word	::=	String
Value	::=	String
	|	Number
	|	AttrValue
	|	‘¡’ ValueList ‘¿’
	|	‘[’ Conjunction ‘]’
	|	‘(’ Disjunction ‘)’
	|	‘#’Number‘(’ Disjunction ‘)’
Word	::=	String
ValueList	::=	ValueList ‘,’ Value
	|	Value

Table 1.4: Lexicon entries (EBNF)

the condition. For all other edges the constraint is automatically fulfilled. The following two constraints are thus equivalent:

```
// JUNK immer unter ROOT
{X:SYN} : syn_junk : 0.0 :
 ~root(X^id) -> X.label!=JUNK;
```

```
// JUNK immer unter ROOT
{X!SYN} : syn_junk : 0.0 :
 X.label!=JUNK;
```

Again, the only difference is that the second constraint is more efficient, since in many cases it is not evaluated at all.

The following table lists all direction indicators:

Indicator	Meaning	Equivalent to
`X|FOO`	X points to NIL	`root(X^id)`
`X!FOO`	X does not point to NIL	`~root(X^id)`
`X/FOO`	X points to the right	`distance(X^id,X@id) < 0`
`X\FOO`	X points to the left	`distance(X^id,X@id) > 0`
`X:FOO`	No restriction	`true`

The connexion between two edges can also be restricted in a similar way. These two constraints are equivalent:

```
// Es gibt nur einen Artikel.
{X:SYN, Y:SYN} : syn_det_zahl : 0.0 :
X^id = Y^id ->
~(X.label=DET & Y.label=DET);
```

```
// Es gibt nur einen Artikel.
{X:SYN/\Y:SYN} : syn_det_zahl : 0.0 :
  ~(X.label=DET & Y.label=DET);
```

The following connexion indicators are available for this purpose:

Indicator	Meaning	Equivalent to
`X:FOO/\Y:BAR`	X and Y have the same regent	`X^id=Y^id`
`X:FOO\/Y:BAR`	X and Y have the same dependent	`X@id=Y@id`
`X:FOO||Y:BAR`	X and Y have no common words	`X^id!=Y^id & X^id!=Y@id & X@id!=Y^id & X@id!=Y@id`
`X:FOO\Y:BAR`	X is below Y	`X^id=Y@id`
`X:FOO/Y:BAR`	X is above Y	`X@id=Y^id`
`X:FOO==Y:BAR`	X and Y are structurally the same	`X^id=Y^id  & X@id=Y@id`
`X:FOO~=Y:BAR`	X and Y are structural inverses	`X^id=Y@id  & X@id=Y^id`
`X:FOO, Y:BAR`	No restriction	`true`

Both the direction and the connexion indicators can also be used in the constraint body, so these two constraints, for example, are equivalent:

```
// Subjekt steht vorn
{X:SYN} : syn_subj : 0.0 :
 X.label = SUBJ -> distance(X^id,X@id) < 0;
```

```
// Subjekt steht vorn
 {X:SYN} : syn_subj : 0.0 :
 X.label = SUBJ -> X/;
```

Again, it is a good idea to use the connexion indicators in the constraint signature rather than in the body where possible.

The next item after the variable declaration is the constraint name. Constraint names must be unique within a grammar, or one of the constraints will be overwritten with a warning.

After the constraint name, a constraint section can be specified. When more than one constraint is member of a section, all of them can be turned off or on at once with the commands 'activate' and 'deactivate'.

The last item before the constraint body is the declaration of its weight. If no weight is specified in the constraint, it is set to a default of 0. The weight can be specified by an expression that is evaluated.

```
// Je weiter entfernt, desto schlimmer
{X/SYN} : syntax : [ 1 / distance( X^id, X@id ) ] :
  X.label = DET ->
  distance( X@id, X^id ) < 3;
```

The constraint body is a logical formula that must evaluate to a Boolean value. The following operators and junctors are available:

Operator	Meaning
`()`	grouping
`~`	logical not
`&`	logical and
`|`	logical or
`->`	logical implication
`<->`	logical biimplication
`+`	arithmetic plus
`-`	arithmetic minus
`*`	arithmetic multiplication
`/`	arithmetic division
`=`	equality
`!=`	inequality
`<`	numeric smaller
`>`	numeric greater
`<=`	numeric smaller or equal
`>=`	numeric greater or equal
`.label`	label of an edge
`.level`	level of an edge

In addition, direction and connexion indicators (see above) can be applied to constraint variables; they then function as boolean expressions.

Equality and inequality are defined both on strings and numbers. The usual evaluation rules apply — times binds stronger than plus, and binds stronger than or, null may not be divided by, etc.

The operators `.label` and `.level` can be applied to constraint variables and return the label or the name of the level of the corresponding dependency edge as a string.

Formulas can evaluate to booleans, numbers, and strings (although the latter two cases are actually 'terms' rather than 'formulas' internally). String and number literals can be written directly without quoting. Boolean literals take the form `true` and `false`.

To access features of the words under consideration, the operators `^` (symbolizing an up arrow) and `@` (symbolizing nothing in particular) are used. The term `X@foo` evaluates to the

value of the feature 'foo' of the word at the lower end of edge `X`. The term `X^foo` does the corresponding thing for the upper word. Thus, the formula `X^case = X@case` postulates case agreement between regent and dependent.

Where features are declared as nested values, they are accessed via the `:` operator. If a grammar sorts its features into syntactic and semantic, the case feature might be accessed as `X@syn:case` or even `X@syn:morph:case`.

Several additional pieces of information about each word can be queried with the same syntax, i.e. they act like features that are automatically defined for each word, but dependent on the token rather than the type.[5] The following pseudo-features exist:

pseudo-feature	meaning
id	identity
word	phonetic form
from	start of timespan
to	end of timespan
info	Verbmobil info field
chunk_start	previous chunk boundary
chunk_end	following chunk boundary
chunk_type	type of current chunk

The identity operation is used to pass a word to functions and predicates that expect lexeme nodes; it is only necessary because `X@` itself is not a valid term.

The 'word' pseudo-feature returns the form of the current word as specified in the lexicon. This may differ from the form that was specified in the lattice because `cdg` tries to undo beginning-of-sentence capitalization and UPPER CAPS EXPRESSIONS when building parse problems. This means that a constraint can safely demand `X@word = und` and be sure that it will always succeed on that word, even if the actual reading was 'Und' or 'UND'.

'from' and 'to' return the start and end points of the time interval that the specified word occupies in the lattice. Currently these time points are always truncated to integers at load time, and they are always the first n integers in lattices that do not specify time information. It is guaranteed that for successive words, one word's 'to' is equal to the next word's 'from'.

'info' returns the Verbmobil comment string if one was specified in the lattice, or causes an error otherwise.

The chunk-related pseudo-features return information that was computed by a chunk parser if one was active while the lexeme graph was created. The exact values returned depend on the chunk parser. In a typical NP, the three return values might be 3, 5, and 'NP'.

When an error of any type occurs during evaluating a formula, evaluation is stopped and the entire constraint fails immediately. Errors can occur

- when applying an operation to formulas that do not have a suitable type
- when accessing the upper word of a NIL dependency edge

[5] There is no sound reason why some accesses are defined as pseudo-features and others as built-in functions; in fact, since the pseudo-features pollute the namespace for real features, a case could be made that all of them should be built-in functions. Feel free to unify the input language.

- when querying a feature that the current word does not have
- upon arithmetic error

All formulas are guaranteed to be evaluated left to right, and short-circuit where possible. Therefore, in the formula `exists(X@case) & X@case = nom` the access to `X@nom` will never generate an error.

1.4.3 Functions

The following functions (FUNCTION in table 1.5) are defined.

- 'abs': Returns the absolute value of the specified number.

```
{X} : 'Adverbs are positioned close to the verb.' : 0.5 :
  X.label=AMOD -> abs( distance( X@id, X^id ) ) < 4;
```

- 'acoustics': Takes a lexeme node and returns the score that the speech recognizer assigned to this reading. (This is the number that can be specified after the time information in the long form of a lattice definition.) If no number is specified there or the lattice has been declared in the short form, the result is always 1.

```
{X} : Recognizer : [ acoustics(X@id) ] :
 acoustics(X@id) = 1.0;
```

- 'distance': Takes two lexeme node labels and returns the distance between them.

```
{X, Y} : 'Subjects come before objects.' :
  X.label=SUBJ & Y.label=OBJ -> distance(X@id, Y@id) > 0;
```

- 'exp': Takes a number x and returns the value e^x.

```
{X!SYN} : 'prefer short edges' : [ exp([ 1 - abs(X@to-X^to) ] / 10) ] :
  abs(X@to-X^to) < 2;
```

- 'height': Takes a lexeme node n and returns the maximal height of a subtree whose root is n.

```
{X!SYN} : 'prefer short edges' : [ exp([ 1 - abs(X@to-X^to) ] / 10) ] :
  abs(X@to-X^to) < 2;
```

- 'lookup': This function takes to or more strings and returns user-defined data. The first string must be the name of a user-defined map. The following strings are combined to form a key into this map, and the corresponding value is returned.

  For instance, German contains many verbs that take separable prefixes, and many different prefixes, but not all verbs take all prefixes. Whether or not a prefix subordination is allowed therefore depends on two separate data items. With a suitable data map (in fact, the one given as an example below), the condition can be expressed like this:

```
{X!SYN} : 'falsches AVZ' : init : 0.0 :
X.label = AVZ
->
exists(X^infinitive) &
lookup(AVZ, X@word, X^infinitive) = ok;
```

- 'match': This function takes three parameters: the name of a defined hierarchy, a value list, and a string. The elements of the value list must be alternating strings and numbers. Each string in the list is checked to see if it subsumes the third parameter. If that is the case, the following number is returned as the result. Otherwise 0 is returned. This function can be used in the following way:

```
sehr := sehr :
[ cat: ADV,
  modifies: <Adjektiv, 1, Verb, 0.9>
];

{X!SYN} : ADV_match : [ match( Kategorien, X@modifies, X^cat ) ] :
  X@cat = ADV
  ->
  match( Kategorien, X@modifies, X^cat ) = 1.0;
```

  The second parameter can, instead of a list, be a single string. In this case the function behaves exactly like the predicate 'subsumes'.

- 'max': Takes two or more numbers and returns the largest.

- 'min': Takes two or more numbers and returns the smallest.

```
{X:SYN, Y:SYN_ND} : np_mod : syn_nd : 0.0 :
  X@id=Y@id & ~root( Y^id ) & ~root( X^id ) ->
  max( Y@to, Y^to ) <= X^from | min( Y@from, Y^from ) >= X^to;
```

- 'parens': This function returns the level of parentheses (introduced by round or square brackets) that a word is in.

- 'parent': This function returns the regent of a given word:

```
{X:SYN,Y:SYN} : Idiom : 0.1 :
  X^word = auf & X@word = Vordermann & Y@word = bringen
  ->
  parent(X^id) = Y@id;
```

  Obviously this is only useful to find the parent of the *regent* of a particular edge `X`; the parent of its *dependent* is always available as `X^id`.

- 'phrasequotes': Similar to 'quotes', this function returns the number of quotation marks that a word is marked by, but it regards only such quotation that is not also marked with commas. (This kind of quotation may be an indicator of nominalization.)

- 'pts': The function name stands for 'POS tag score'. The function takes a lexeme node and returns the score that the POS tagger assigned to this lexical reading. This is *not* declared in the input; a score other than 1 occurs only if a POS tagger was actually active when the lexeme graph was built. The usual way to use this score is to demand that it should be as high as possible. Here's how to integrate the POS tag score into a grammar with a cutoff of 0.1:

```
{X:SYN} : tagger : [ min(0.1, pts(X@id)) ] :
  pts(X@id) = 1.0;
```

- 'quotes': This function returns the number of quotation marks that the word is nested into. This will usually be 0 or 1.

  The counting is only approximately accurate for various reasons:

  – If an entire paragraph of text is quoted, but is modelled as individual word graphs, the second and following sentences will have no quoting information associated with them, since the computation only regards one word graph at a time.
  – When sentence and quotation boundaries differ, opening and closing quotations may be confused. However, neighbouring punctuation is used as a heuristic that usually chooses the right alternative.

- 'realcat': This function takes a lexeme node and returns the correct POS tag. (This is determined by reading the corresponding annotation.) Its only use is to simulate parsing with a perfect POS tagger.

1.4.4 Predicates

Predicates differ from functions only in that they return Boolean values rather than strings or numbers. The following predicates (PREDICATE in table 1.5) are defined:

- 'between': Takes two lexeme nodes and a string and returns `TRUE` if the two words are separated by at least one instance of a punctuation character contained in the string.

```
{X!SYN} : 'comma needed for subclauses' : 0.1 :
X.label = SUBC
->
between(X@id,X^id,",");
```

- 'chunk_head': Takes a lexeme node and returns `TRUE` if chunk information is present and the chunk parser designated this word as the head of its chunk.
- 'compatible': Takes three parameters: the name of one hierarchy and two types. It returns 'true' if one of the types subsumes the other in the hierarchy.
- 'connected': Takes two lexeme nodes and returns `TRUE` if they are both part of the same tree.
- 'cyclic': Takes the name of a level and returns `TRUE` if the dependency structure on that level is cyclical.
- 'exists': Takes one parameter and checks if it contains a valid combination of attribute values. Because of short-circuit evaluation, this function can be used to avoid errors due to missing features:

```
{X} : 'Subjects are nominative' :
  X.label = subject
  ->
  exists(X@case) & X@case = nom;
```

- 'has': takes a lexeme node identification and a string. It then determines whether on the 'current' level (the level of the edge used to access the word), at least one edge with the specified label exists that modifies this node.

  This constraint posits that any noun has a determiner:

  ```
  {X:SYN} : 'Nouns have determiners' :
    X@cat = NN -> has(X@id,DET);
  ```

  The second argument may also be the name of a unary constraint; in this case, the constraint is called on each candidate edge and the `has` succeeds if it succeeds at least once. (This iteration short-circuits like a logical and, i.e. after one application succeeds, the helper constraint is not called again.)[6]

  If there is a third argument, it must be the name of a hierarchy which should be used for label matching. Thus, a constraint can say '`has(X@id,OBJA_OBJC,Labels)`', and (assuming a fitting 'Labels' hierarchy) either `OBJA` or `OBJC` will satisfy the predicate. This allows several calls to the simple form of 'has' to be merged into one.

  If there is a fourth argument, the search is conducted recursively. The value of the argument is a node in the previously specified hierarchy. This node subsumes all the labels which are *allowed* on the path through the tree. So to express that anything which has a relative pronoun in its scope is a relative clause, you can say

  ```
  X@cat = vfin & has(X@id, find_prel, Labels, AUX)
  ->
  X.label = REL
  ```

  This constraint forces a verb to be a relative clause if one word in its VP has a relative pronoun as a subordinate, but not if verb in a subordinated clause has a relative pronoun, because then at least one label other than `AUX` will intervene.

  If there is a fifth and sixth argument, they should be numbers and will then constrain the range of time points within which a fitting edge will be searched.

  Using this predicate in a constraint has far-reaching consequences. It turns *local* constraints into *global* constraints, which in some circumstances are more expensive to evaluate. It is included because the only alternative way of providing this important functionality is writing auxiliary levels which on the whole are just as expensive, but less intuitive for modeling.

- 'is': takes a lexeme node identification and a string. It then determines whether on the 'current' level (the level of the edge used to access the lexeme node), the label of this node is the specified string or not.

  ```
  {X/SYN/\Y/SYN} : Vorfeld :
    X^cat = VVFIN -> ~is(X^id,ROOT);
  ```

[6] Obviously the second version is allows everything the first does and much more. The first version is retained both for backward compatibility and because it is clearer to read in simple cases.

It is not useful to use `is` on an expression such as `X@id`, since the label of `X` is already accessible as `X.label`; but by using it on `X^id`, the label of the unnamed (but unique) edge above `X` and `Y` can be checked.

If there is a third argument, it is used as the name of the level on which the subordination should be checked. If there is none, the level that is first named in the constraint signature is used.

Formally, use of the predicate 'is' turns a local constraint into a global constraint just like 'has' does, but since the condition that it represents can be checked in constant rather than linear time, it is usually no more expensive than a normal local constraint.

- 'nonspec': Takes a lexeme node specification and returns `TRUE` if it resolves to a NIL binding or to an explicitly underspecified binding.
- 'occur': Takes a string as a parameter and checks whether this word form occurs anywhere in the current sentence.
- 'print': Takes any number of parameters (numbers and strings), prints them on the standard output, and returns 'true.'

```
{X} : Ausgabebeispiel :
  X@cat=Verb ->
    root( X^id ) &
    print( 'Hurra, Wurzel gefunden: ', X@word ) &
    print( ' von ', X@from, ' bis ', X@to );
```

- 'root': Takes one lexeme node identification and returns 'true' if it is the root node.

```
{X} : 'Verben an die Wurzel' :
  X@cat=Verb -> root( X^id );
```

- 'spec': The opposite of 'nonspec'.
- 'start': Takes one lexeme node and returns 'true' if it is at the beginning of a word graph (i.e. if it starts at the earliest possible time point).
- 'stop': Takes one lexeme node and returns 'true' if it is at the end of a word graph (i.e. if it ends at the latest possible time point).
- 'subsumes': Takes three parameters: the name of a defined hierarchy and two character strings. It returns 'true' if the first string subsumes the second in the hierarchy.

```
ontology -> top( animate( human, animal ), inanimate );

{X} : 'Subjekt von sehen ist belebt.' :
  X^word=sehen & X.label=SUBJ ->
    subsumes( ontology, animate, X@semtype );
```

- 'under': Takes two lexeme nodes and returns 'true' if the first word is directly or indirectly subordinated under the second one.

1.4.5 Lexicon entries

Lexicon entries have the following formats:

```
auto :=
  [ cat:noun, num:sg,
    prop:thing
  ];

'O Neal' :=
  [ syn: [ cat:name,
           subcat:none,
           agr:'3sg'
         ],
    sem:foo:bar:baz
  ];

sich :=
 [ syn:[ cat:pron, num:(sg|pl), pers:3, case:(dat|acc) ],
   sem:[ cat:NIL ]
 ];
```

The lexicon entries do *not* contain full feature structures, as in unification based grammars. Co-reference and unification is not supported.

Embedded items in the feature structure can be referred to by giving all the attributes. The expression `X@syn:cat`, for example, if `X@` points to the lexeme `sich` as above, returns the value `pron`. Each attribute can only occur once in the structure.

Lists of values (enclosed by '¡' and '¿') can only be accessed with the function 'match' and not directly (cf. section 1.4.3).

Disjunctions are equivalent to writing a separate lexicon entry for each combination of values. The last example above is thus an abbreviation for four different entries for 'sich.' Disjunctions can be recursively embedded:

```
der :=
 [ syn:[ cat:def_article,
        ([ num:pl, gen:(mas|fem|neu), case:gen ] |
         [ num:sg, ([ gen:mas, case:nom ] |
                    [ gen:fem, case:(gen|dat) ]) ])
       ],
   sem:[ cat:NIL ]
 ];
```

Disjunctions can be coupled together using indexes.

```
Tor :=
 [ syn:[ cat:noun, gen: #1(neu|mas) ],
   sem:[ cat: #1(BUILDING|PERSON) ]
 ];
```

The condition above has two cases, not four. There is one case with `syn:gen = neu` and `sem:cat = BUILDING`; one with `syn:gen = mas` and `sem:cat = PERSON`. In the current implementation only index numbers 1–9 are supported.

When multiple lexicon entries describe the same orthographic form, they will be given automatically generated labels. These labels are printed on screen if the command `set debug on` has been given.

```
INFO: grapheme graph: #nodes 1, min 0, max 1
treffen ===> treffen_ø
treffen ===> treffen_1
treffen ===> treffen_3
INFO: lexem graph: #nodes 3, min 0, max 1, #paths 3
```

In this example, the lexicon contains three entries for the word 'treffen.' The automatic renaming suggests that one stands in the first person, one in the third person, and that for one the person is unspecified. Which features are preferably used in this renaming step can be influenced by setting the CDG variable `anno-categories`.

It is also possible to write *templates* that generate lexical entries for unknown words as needed. Instead of giving the exact form of a word, only a *regular expression* is given with the operator `=~`. If an unknown word is encountered that matches the template, a lexical entry is generated automatically as if it had been there all along. Here is how to provide for all possible arabic numerals with only one lexical template:

```
'^[+-]?[0-9]+$' =~
 [ cat:CARD,
   case:bot,
   person:third,
   number:pl,
   gender:bot,
   sort:number ];
```

Use of templates is governed by the setting of the 'templates' variable.

In addition to the template mechanism, `cdg` has a few more tricks up its sleeve when finding lexical entries. First of all, when a capitalized word cannot be found and circumstances lead it to suspect (such as at the beginning of a sentence) that its citation form is actually lower case, the lower case form is looked up instead. Likewise, UPPER-CAPS words are tried in their lower case versions if they are not found. Furthermore, words from the categories named in 'capitalizable-categories' can be found even if they are actually capitalized in the input.

Also, lexical entries can be accessed directly from disk without loading the entire list into RAM. If the 'uselexicon' command is given, lexical entries are also looked up in the specified database. This avoids huge memory consumption when using a real-word lexicon. The external data base and its index must have been generated properly with the `indexer` command available in the `cdg` package.

Finally, if `deduceCompounds` is set, `cdg` tries to detect explicitly compounded words. Then, if there is no entry for 'T-Aktie' but one for 'Aktie', a lexicon entry for 'T-Aktie' will be

auto-generated when the need arises. The new entry has all the features of the old one except that its phonetic form is different.[7]

Unmarked compounds such as 'Stammaktie' can also be detected if `compound-categories` is set. It must be a comma-separated list of categories that are expected to form unmarked compounds. For the STTS tagset a value might be 'NN,ADJA,ADJD'.

1.4.6 Hierarchies

The term 'hierarchy' is not accurate. Both genuine hierarchies (trees) and directed acyclic graphs can be defined.

```
types ->
 top( concrete( animate(animal, plant), inanimate ),
      abstract( concept, event )
 );

Verben ->
 Verb -> Vollverb Auxiliarverb Modalverb,
 Verb -> finit infinit Partizip,

 INF    <- Vollverb infinit,
 FIN    <- Vollverb finit,
 PPP    <- Vollverb Partizip,

 VMODINF   <- Modalverb infinit,
 VMODFIN   <- Modalverb finit,
 VMODPPP   <- Modalverb Partizip,

 VAUXINF   <- Auxiliarverb infinit,
 VAUXFIN   <- Auxiliarverb finit,
 VAUXPPP   <- Auxiliarverb Partizip
;
```

The two nodes 'top' and 'bot' are pre-defined as the top and bottom nodes in each hierarchy. Therefore 'top' can not be subsumed by any other node, and 'bot' can not subsume another.

1.4.7 Data maps

Data maps are defined with the `=>` operator. They can be accessed with the `lookup()` function. Here is the beginning of a map that specifies the valence frames depending on the base verb and prefix:

```
AVZ =>
  ab arbeiten       => A,
  auf arbeiten      => A,
  aus arbeiten      => A,
  ein arbeiten      => A,
```

[7] Note that this kind of semantics is appropriate for German but not necessarily for other languages.

```
  heraus arbeiten   => AC,
  mit arbeiten      => '-',
  nach arbeiten     => 'A?',
  weiter arbeiten   => '-',
  zusammen arbeiten => '-',
  ...
;
```

1.4.8 Word graphs

Here come two possible word graphs. The format is very similar to that used in VERBMOBIL. Conversion between the two formats is trivial.

```
'M/1/1 Kette' :
  pferde,
  fressen,
  gras;

'M/1/1 Graph' :
  0 1 pferde    0.1,
  0 1 er        0.05,
  1 2 fressen   0.3,
  1 2 befestigt 0.1,
  2 3 gras      0.4;
```

The word graphs, of which there can be more than one per sentence, have the labels '`M/1/1 Kette`' and '`M/1/1 Graph`'.

1.4.9 Annotations

Annotations are the manually specified correct analyses for the inputs. The following example shows an utterance with syntactic categories and syntactic level dependencies.

```
'M/1/1' : 'M/1/1' <->
  0 1 pferde
      cat/noun
      SYN->SUBJ->2
      SEM->AG->2,
  1 2 fressen
      cat/verb
      SYN->ROOT->0
      SEM->ROOT->0,
  2 3 gras
      cat/noun
      SYN->OBJ->2
      SEM->PAT->2
;
```

The first identifier is the name of the annotation, the second is the name of the lattice that it refers to. There can be more than one annotation for the same lattice.

Is important that the annotations and the word graphs fit each other exactly, i.e. that both the start and end points, and the word forms are the same. Otherwise no comparison is possible.

Attr	::=	String
	|	Number
Connexion	::=	',' |'/\' |'\/' |'——' |'==' |'~=' |'\' |'/'
ConstraintId	::=	String
Constraint	::=	'{' VarList '}' ':'
		ConstraintId ':' [[Section] ':' Penalty ':']
		Formula
Direction	::=	':' |'—' |'\' |'/' |'!' |
Formula	::=	Term Relation Term
	|	Formula Junctor Formula
	|	Predicate '(' TermList ')'
	|	Variable Connexion Variable
	|	Variable Direction
	|	'~' Formula
	|	'(' Formula ')'
	|	'true'
	|	'false'
Function	::=	String
Junctor	::=	'&' |'|' |'->' |'<->'
Operator	::=	'+' |'–' |'*' |'/'
Path	::=	Path ':' Attr
	|	Attr
Penalty	::=	Number
	|	Term
Predicate	::=	String
Relation	::=	'=' |'>' |'<' |'>=' |'<=' |'!='
Section	::=	String
Term	::=	Variable '^' Path
	|	Variable '@' Path
	|	Variable '.label'
	|	Variable '.level'
	|	Term Operator Term
	|	Function '(' TermList ')'
	|	'[' Term ']'
	|	String
	|	Number
TermList	::=	Term
	|	TermList ',' Term
Variable	::=	String
VarInfo	::=	Variable Direction String
VarList	::=	VarInfo
	|	VarInfo Connexion VarInfo

Table 1.5: Constraints (EBNF)

AnnoId	::=	String
Arc	::=	Stars From To Word [Penalty]
	|	Stars Word
ArcList	::=	ArcList ',' Arc
	|	Arc
From	::=	Number
Penalty	::=	Number
Stars	::=	'*' [Stars]
To	::=	Number
Word	::=	String
WordGraph	::=	WordGraphId ':' AnnoId ':' ArcList
WordGraphId	::=	String

Table 1.6: Word graphs (EBNF)

AnnoId	::=	String
Annotation	::=	From To Word SpecSeq
AnnoEntry	::=	AnnoId '¡-¿' AnnoList
AnnoList	::=	AnnoList ',' Annotation
	|	Annotation
DepKind	::=	String
From	::=	Number
Label	::=	String
Position	::=	Number
Specification	::=	TagKind '/' TagName
	|	DepKind '->' Label '->' Position
SpecSeq	::=	*empty*
	|	SpecSeq Specification
TagKind	::=	String
TagName	::=	String
To	::=	Number
Word	::=	String

Table 1.7: Annotations (EBNF)

Hierarchy	::=	Id '− >' Sort
	|	Id '− >' SubsumptionList
Sort	::=	String '(' SortList ')'
	|	String
SortList	::=	Sort
	|	SortList ',' Sort
SubsumptionList	::=	SubsumptionList ',' Subsumption
	|	Subsumption
Subsumption	::=	String '− >' TypeSeq
	|	String '< −' TypeSeq
TypeSeq	::=	TypeSeq String
	|	String

Table 1.8: Hierarchy definitions (EBNF)

Chapter 2

Der graphische Parser `Xcdg`

2.1 Allgemeines

`Xcdg` ist eine graphische Benutzeroberfläche für den `cdg`-Parser. Es bietet eine Obermenge der bisherigen Funktionalität. Zusätzliche Möglichkeiten sind

- Ausführung von Kommandodateien in der `cdg`-Shell-Sprache
- Anwendung von Prädikaten auf mehrere Argumente durch "select and click"
- nach Ebenen gruppierte Darstellung der Lösungen
- graphische Darstellung, Manipulation und Ausdruck von Stemmata
- Bewertung eigener Analysen

2.2 Installation

Um `Xcdg` zu installieren, reicht es gewöhnlich aus, im Verzeichnis `cdg` den Befehl

```
make install
```

aufzurufen und die Anweisung

```
Please set your XCDG_PATH to <Pfadname>.
```

zu befolgen. Ebenso gut kann das Programm aber auch aus dem Quellverzeichnis aus gestartet werden.

2.3 Aufruf

`Xcdg` wird aus der Shell unter seinem eigenen Namen aufgerufen:

```
$ xcdg.tcl &
```

Es akzeptiert folgende Optionen auf der Kommandozeile:

- `-e <filename>` gibt eine Kommandodatei an, die beim Start gelesen wird.
- `-grammarpath <path>` setzt den Pfad, in dem zu ladende Grammatiken gesucht werden.
- `-noinit` verhindert, daß die Datei `~/.xcdgrc` gelesen wird.

Das Programm liest zuerst die globalen X11–Ressourcen aus der Datei Cdgrc, die zuerst im Verzeichnis `$XCDG_PATH` und danach im aktuellen Verzeichnis gesucht wird.

Für das `Xcdg`-Hauptfenster sind zum Beispiel folgende Werte vordefiniert:

```
*CdgMain.height: 800
*CdgMain.width: 600
*CdgMain.TabPos: w
*CdgMain.grammarPath: .
```

Anschließend wird die Datei `.xcdgrc` gelesen und ausgeführt. Dabei handelt es sich um eine Kommandodatei, in der Befehle gesammelt werden können, die auch auf der Befehlszeile ausgeführt werden können. Diese Befehle werden also bei jedem Programmstart ausgeführt. Ein Benutzer, der jederzeit maximale Information über den Verarbeitungszustand erhalten will, kann zum Beispiel diese Zeilen in sein `.xcdgrc` schreiben:

```
set subsumesWarnings full
set verbosity on
set showdeleted on
```

Diese Datei wird zunächst im aktuellen Verzeichnis gesucht, dann im Heimverzeichnis des Benutzers.

Letztlich werden alle Kommandodateien, die mit der Option `-e` angegeben wurden, ausgeführt.

2.4 Das `Xcdg`-Fenster

Das `Xcdg`-Fenster (siehe Abb. 2.1) besteht aus einer Menüleiste, der Anzeigebox und dem `cdg`-Shell-Fenster. Die Anzeigebox und das Shell-Fenster teilen sich einen Viewport. Mit dem quadratischen Button zwischen ihnen kann der Anteil beider Regionen eingestellt werden. Mithilfe des Menüs 'Windows' können die beiden Regionen ein- und ausgeblendet werden. Im unteren Bereich des `Xcdg`-Fensters befindet sich eine Hilfezeile, in der zu jedem Objekt der Anwendung (Buttons, Menüs, Dateinamen etc.) eine Information angezeigt wird.

2.5 Die Menüleiste

Die Menüleiste wird entweder mit der Maus bedient oder auf die übliche Weise mit den Tasten F10, ESC, den Cursortasten und Return. Die hervorgehobenen Buchstaben können auch durch die Kombination mit der Alt-Taste aufgerufen werden.

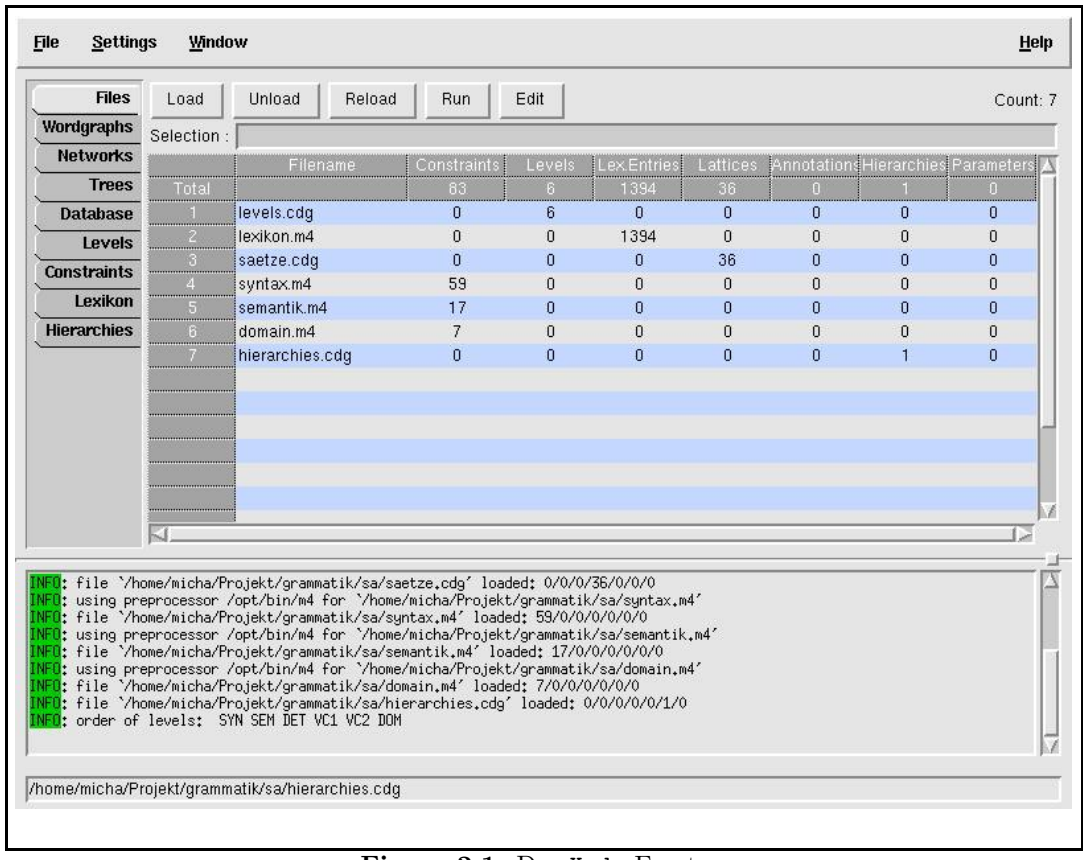


Figure 2.1: Das `Xcdg`-Fenster

2.5.1 Das Menü 'File'

Der Menüeintrag 'Load' öffnet den Dateiauswahldialog. Durch ihn kann eine Datei in den Parser geladen werden. Die Wirkung ist, als wäre der betreffende Dateiname in der `cdg`-Shell als Argument zum Befehl `load` angegeben worden.

- Der Button 'Directory' erlaubt es, durch Ziehen der Maus schnell jedes Verzeichnis über dem aktuellen Verzeichnis auszuwählen.
- Mit dem Icon 'Folder up' kann um genau ein Verzeichnis in der Hierarchie nach oben gewechselt werden.
- Im Dateinamen-Fenster wählt ein Mausklick die betreffende Datei aus, ein Doppelklick lädt sie sofort.
- Im Fenster 'File name' erscheint der selektierte oder eingetippte Dateiname.
- Der Button 'File of type' bestimmt, welche Dateinamen angezeigt werden. Diese Maske erlaubt normalerweise alle Dateien, die normale `cdg`-Eingaben enthalten, deren Namen also auf `.cd?` oder `.m4` enden. Stattdessen können auch nur Dateien mit einzelnen Grammatikfragmenten angezeigt werden (etwa die auf `.cdl` endenden), alle `cdg`-Skriptdateien (auf `.run` oder `.scr`) oder alle Dateien. Die Buttons 'Open' und 'Cancel' schließlich bestätigen oder verwerfen die Auswahl.

Der Menüeintrag 'Run' funktioniert wie der Eintrag 'Load', aber er lädt keine cdg-Datei, sondern eine Kommandodatei, deren Befehle sogleich ausgeführt werden. Die Wirkung ist, als wäre der Inhalt der Datei zeilenweise in die cdg-Shell eingegeben worden. Die empfohlene Endung für die Namen von Kommandodateien ist `.scr`. Daher ist auch die Maske `*.run,*.scr,*.cdg` voreingestellt.

Der Menüeintrag 'Preferences' läßt den Benutzer die Angaben über persönliche Vorlieben editieren:

- Benutzername
- Eingabepfad für verwendete Grammatiken
- Aufrufsyntax des bevorzugten Editors

Der Menüeintrag 'Reset' entspricht dem CDG-Befehl 'reset'.

Der Menüeintrag 'Quit' bricht das Programm ab.

2.5.2 Das Menü 'Settings'

Im Menü 'Settings' können verschiedene programmglobale Flags eingestellt werden. Die Wirkung ist, als wäre das betreffende cdg-Flag mit `set` (siehe Abschnitt 1.2.41) eingestellt worden.

Das zusätzliche Flag 'Auto-display' kommt bei approximativen Lösungsverfahren zur Anwendung; wird es gesetzt, so werden neu gefundene Analysen ohne Aufruf von 'showparse' graphisch dargestellt (diese Einstellung kann die Analyse übermäßig verlangsamen).

2.5.3 Das Menü 'Windows'

In diesem Menü kann die Anzeige der beiden Hauptkomponenten Shell und Anzeigebox gesteuert werden. Wird nur eine der beiden Darstellungen angezeigt, so nimmt sie das gesamte X-Fenster ein.

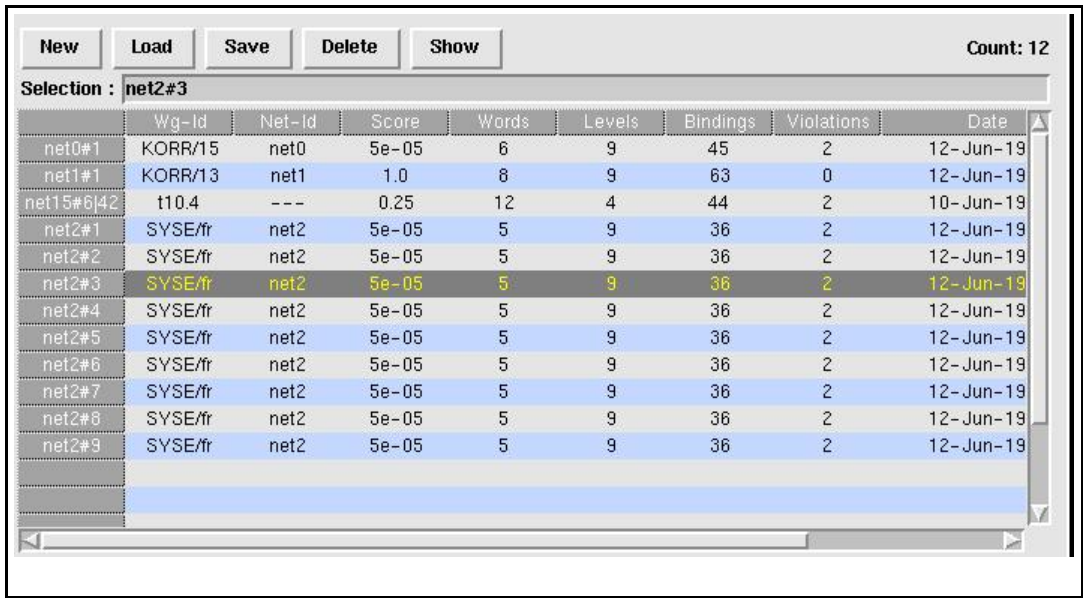
2.5.4 Das Menü 'Help'

Der Menüeintrag 'About' gibt Information über die Herkunft des Programms. Der Menüeintrag 'Help' startet die Online-Hilfe, die sich aber auf einen Hinweis auf diesen Text beschränkt.

2.6 Die Anzeigebox

In der Anzeigebox werden alle vorhandenen Daten des Parsers dargestellt (siehe Abb. 2.2). Sie besteht aus

- dem *Register* zur Auswahl des Datentyps. Ein Linksklick auf eine Registerzunge öffnet das betreffende Register.



	Wg-Id	Net-Id	Score	Words	Levels	Bindings	Violations	Date
net0#1	KORR/15	net0	5e-05	6	9	45	2	12-Jun-19
net1#1	KORR/13	net1	1.0	8	9	63	0	12-Jun-19
net15#6|42	t10.4	---	0.25	12	4	44	2	10-Jun-19
net2#1	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#2	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#3	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#4	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#5	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#6	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#7	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#8	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19
net2#9	SYSE/fr	net2	5e-05	5	9	36	2	12-Jun-19

Figure 2.2: Die Anzeigebox

- einer Reihe von *Buttons* zur schnellen Ausführung von Befehlen. Ein Linksklick auf einen Button führt den betreffenden Befehl aus.
- dem *Zähler*, der angibt, wieviele Datenstrukturen geladen sind
- der *Selektion* zur Angabe von Argumenten für die Buttons. In der Selektion stehen jederzeit die Namen aller selektierten Datenstrukturen. Klickt man auf die Selektion, so kann sie als normaler Text editiert werden. (Widersprechen sich die Darstellungen von Selektion und der Tabelle, so hat die Selektion recht.)
- der *Tabelle* geladener Datenstrukturen. In ihr sind die geladenen Daten alphabetisch nach ihren Bezeichnern geordnet dargestellt. In den Spalten der Tabelle sind die Werte jeder Struktur angegeben.

Ein Linksklick in der Tabelle selektiert die Struktur in der betreffenden Zeile. Durch Drücken der Control-Taste kann mit einem weiteren Linksklick eine Zeile selektiert werden, die die bisherige Selektion erweitert. Ein Linksklick mit gedrückter Shift-Taste selektiert die Zeilen von der zuvor selektierten Zeile bis zur selektierten Zeile einschließlich. Ein Linksklick auf das linke obere Feld der Tabelle schließlich selektiert alle Zeilen. Die Selektion wird aufgehoben, indem man auf eine leere Zeile klickt.

Im allgemeinen führt der Druck auf einen Button dazu, daß der betreffende Befehl nacheinander auf alle selektierten Daten angewendet wird, so als wäre er so oft wie nötig in die `cdg`-Shell eingegeben worden. Mit Buttons und mehrfacher Selektion können also alle Befehle auf mehrere Datenstrukturen angewendet werden, auch wenn sie in der `cdg`-Shell nur ein Argument verarbeiten können.

2.6.1 Das Register 'Files'

Im Register 'Files' werden alle geladenen Dateien angezeigt. Zu jeder Datei ist angegeben, wieviele Constraints, Ebenen etc. sie definiert. Als Bezeichner für Dateien wird der bloße Dateiname ohne Pfadangabe verwendet. Der volle Dateiname erscheint in der Hilfezeile, wenn sich der Mauszeiger über der entsprechenden Zeile befindet.

Die Buttons 'Load', 'Run' und 'Reset' öffnen denselben Dialog wie die Menüeinträge Files:Load, Files:Run und Files:Reset. Sie beziehen sich also ausnahmshalber nicht auf die Selektion, sondern erfragen den Dateinamen interaktiv.

Der Button 'Edit' öffnet die selektierten Dateien mit dem Editor, der unter File:Preferences: Editor eingestellt ist.

2.6.2 Das Register 'Wordgraphs'

Im Register 'Wordgraphs' werden alle geladenen Wortgraphen angezeigt. Zu jedem Wortgraphen ist die Identifikation und der Wortlaut angegeben.

Der Button 'New' läßt den Benutzer einen neuen Wortgraphen erzeugen. Er ist noch nicht implementiert. Ein linearer Wortgraph ohne Straffaktoren kann durch den Shell-Befehl `inputwordgraph` (siehe Abschnitt 1.2.19) erzeugt werden.

Der Button 'Details' wirkt sich nur auf einen Wortgraphen aus, egal wieviele selektiert sind. Im ausgewählten Wortgraphen werden Anfangs– und Endpunkt sowie der Score und die Wortform jeder Kante angezeigt, was die Struktur von nichtlinearen Wortgraphen besser erkennen läßt als die bloße Aufzählung der Lautungen. Die Darstellung kann nicht editiert werden.

Der Button 'New Net' führt den Befehl `newnet` auf die selektierten Wortgraphen aus. In einem Dialog kann zusätzlich zu den gewählten Wortgraphen noch der Wert der Variable `unaryFraction` für diesen Vorgang eingestellt werden. Wenn mehrere Wortgraphen selektiert sind, so erhalten die Constraintnetzwerke aufeinanderfolgende Nummern. Neu erzeugte Constraintnetzwerke werden sogleich ins Register Networks aufgenommen.

Der Button 'Annotation' sucht in der geladenen Grammatik nach einer Annotation für den selektierten Wortgraphen. Wird eine Annotation gefunden, so wird sie ins Register Parses aufgenommen. (Wenn die Option Auto-Display gewählt ist, wird die Analyse außerdem sogleich graphisch dargestellt.)

Der Button 'IC' wendet den Befehl 'incrementalcompletion' auf den selektierten Wortgraphen an. Auch hier können Zwischenergebnisse automatisch angezeigt werden.

Der Button 'Interactive' startet den Befehl 'incrementalcompletion' im interaktiven Modus. Dabei wird ein vom Benutzer eingetippter Satz sogleich inkrementell analysiert. (Unbekannte Worte bringen die Verarbeitung allerdings zum Abbruch.)

2.6.3 Das Register 'Networks'

Im Register 'Networks' werden alle geladenen Constraintnetzwerke angezeigt. Insbesondere wird zu jedem Netz angezeigt, wieviele Lösungen es hat und welche Bewertungen sie haben.

Der Button 'Delete' führt den Befehl `netdelete` auf seine Argumente aus. Wenn ein Netz gelöscht wird, können seine Lösungen in Form von Bäumen dennoch im Speicher verbleiben.

Die Buttons 'Arcconsistency' und 'Pruning' führen die entsprechenden Befehle auf ihre Argumente aus. Weitere Argumente als der Netzname können nicht übergeben werden. Ein Befehl wie etwa `pruning net0 quadratic` kann nur in der Shell eingegeben werden.

Der Button 'Search' führt den Befehl `netsearch` auf die selektierten Netzwerke aus. In einem Dialog kann außer den Namen der Netzwerke die Suchmethode, die maximale Agendagröße und die Bewertungsschwelle für den Suchraum eingestellt werden.

Die Buttons 'Frobbing' und 'Gls' führen ebenfalls die gleichnamigen Befehle auf die selektierten Netze aus. Genauere Parameter können nur in der Shell angegeben werden.

2.6.4 Das Register 'Parses'

Im Register 'Parses' werden die Analysen dargestellt, die durch Suche in Constraintnetzwerken gefunden wurden. Jede Lösung, die in einem Constraintnetzwerk gefunden wurde, ist eine Schar von so vielen Stemmata, wie Beschreibungsebenen verwendet wurden. Zu jeder Lösung sind verschiedene statistische Daten verfügbar. Als Bezeichner für eine Lösung wird der Name ihres Constraintnetzwerkes und eine laufende Nummer verwendet.

Der Button 'Delete' löscht eine oder mehrere Lösung aus dem Speicher. Alle in einem Lösungsfenster dargestellten Bäume werden ebenso entfernt. Besitzt ein Lösungsfenster keine Darstellungen mehr, wird es geschlossen.

Der Button 'Tree' stellt eine oder mehrere Lösungen graphisch dar. Existiert bereits ein Lösungsfenster, in dem andere Lösungen desselben Netzes dargestellt werden, so werden diese in einem Fenster akkumuliert. Bereits dargestellte Bäume können nicht zweimal dargestellt werden. Wurden mehrere Lösungen verschiedener Netze selektiert, so werden sie in Abhängigkeit ihrer Netzzugehörigkeit auf verschiedene Lösungsfenster verteilt.

2.6.5 Das Lösungsfenster

Die Stemmata der Lösungen werden in einem eigenen Fenster angezeigt (siehe Abb. 2.3).

Das untere Indexregister wählt eine von mehreren Lösungen aus, wenn mehrere Lösungen angezeigt werden. Das obere Ebenenregister wählt eine Ebene der Lösung zur graphischen Darstellung aus. Verstößt die Lösung gegen Constraints, so ist der Name der betreffenden Ebene rot hervorgehoben.

Im Lösungsfenster selbst wird jeweils ein Stemma einer Lösung dargestellt. Über dem Baum wird sein Bezeichner und seine Bewertung angezeigt. Die Lexeme der Lösung werden durch ihre Bezeichner dargestellt. Das Stemma stellt die Unterordnungen der Lexeme durch Kanten und die Labels durch Beschriftung der Kanten dar. Unterhalb des Stemmas der Haupt-Analyseebene sind die nichttrivialen Kanten der übrigen Ebenen durch Pfeile angedeutet.

Verstößt das dargestellte Stemma gegen Constraints, so sind diese unter dem Stemma angeführt. Angezeigt werden Verletzungen und nicht Constraints; ein Constraintbezeichner kann also mehrfach auftreten.

Wird eine angezeigte Constraintverletzung vom Mauszeiger berührt, so werden die betreffenden Kanten und Label rot hervorgehoben. (Kante und Label werden dabei als Einheit

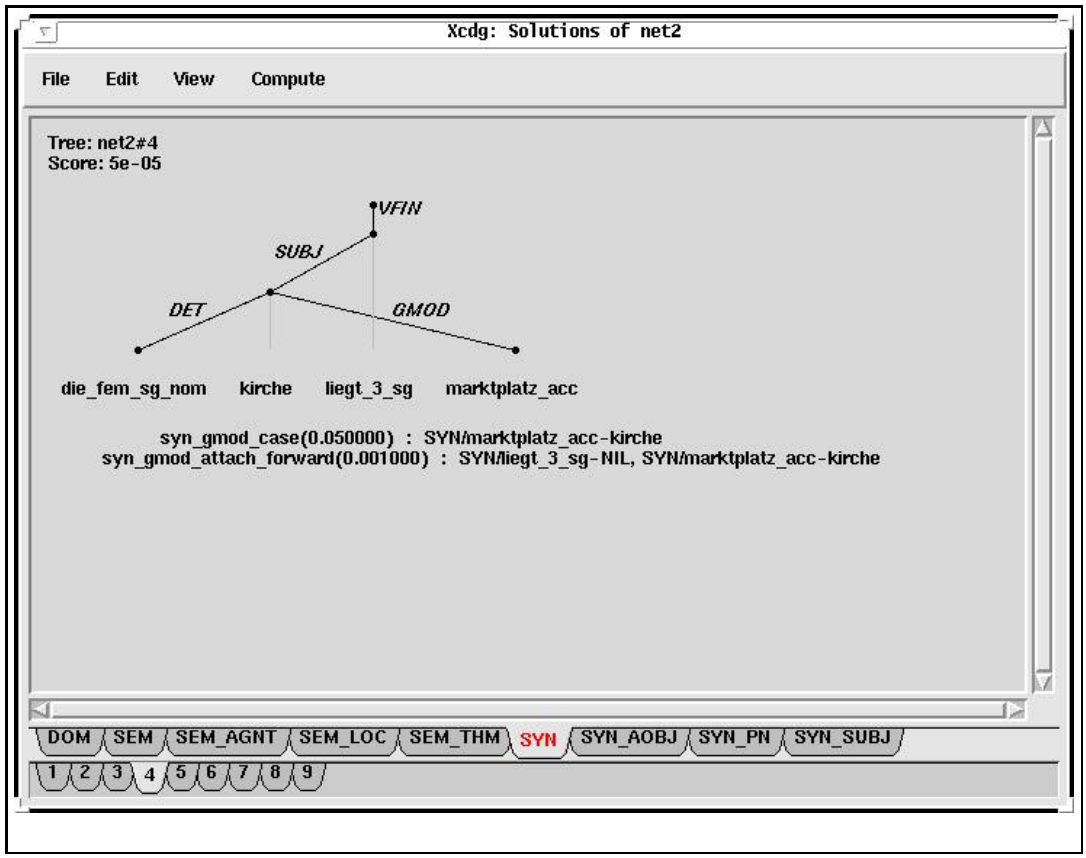


Figure 2.3: Das Lösungsfenster

behandelt, es werden also beide hervorgehoben, auch wenn das Constraint in Wirklichkeit nur gegen das Label oder nur gegen die Unterordnung etwas einzuwenden hat.)

Die angezeigte Lösung kann auf mehrere Arten manipuliert werden:

- Durch einen Linksklick auf ein Lexem öffnet sich eine Liste aller bekannten Homonyme, aus denen ein anderes Lexem ausgewählt und in die Lösung eingesetzt werden kann. Diese Veränderung betrifft alle Stemmata der Lösung, nicht nur das angezeigte.
- Eine Unterordnung kann geändert werden, wenn man die betreffende Kante mit der Maus auf einen anderen Knoten zieht. Ein Lexem kann NIL untergeordnet werden, wenn man seine Kante auf den Wurzelknoten zieht. Zyklische Unterordnung von Kanten erzeugt eine Fehlermeldung. Wenn ein Stemma von Hand verändert wird, werden weiter die Constraintverletzungen des Ausgangszustandes angezeigt, die möglicherweise sinnlos geworden sind. Mit dem Menüpunkt
- Durch einen Linksklick auf ein Label öffnet sich eine Liste mit alternativen Labeln der Ebene, von denen eines ausgewählt werden kann.

(Die Pfeile zur Darstellung weiterer Ebenen sind nicht editierbar.)

Der Menüeintrag ‘Quit’ schließt das Lösungsfenster. Es erfolgt keine Nachfrage, auch wenn veränderte und ungesicherte Lösungen existieren.

Das Menü ‘Edit’ dient der Manipulation des angezeigten Stemmas. Der Eintrag Undo macht jeweils die letzte von Hand vorgenommene Veränderung am Stemma rückgängig, also die Veränderung einer Unterordnung oder eines Labels. Das Austauschen von Lexemen kann nicht automatisch rückgängig gemacht werden.

2.6.6 Das Register ‘Levels’

Das Register ‘Levels’ stellt alle geladenen Ebenendeklarationen dar. Sie werden alphabetisch geordnet aufgelistet. Zu jeder Ebene ist die definierende Datei, die Zahl der Labels, die Zahl der betrachteten Features, die Zahl der unären und binären Constraints, der Status und die durchschnittliche Bewertung der Constraints angegeben.

Der Button ‘Display’ zeigt die selektierten Ebenen in der Shell an, so als wäre dort der Befehl `showlevel` verwendet worden.

Der Button ‘Show’ schaltet die Darstellung der selektierten Ebenen an oder aus, so als wäre in der Shell der Befehl `showlevel` angewendet worden.

Der Button ‘Use’ schaltet die selektierten Ebenen an oder aus, so als wäre in der Shell der Befehl `uselevel` angewendet worden.

Der Button ‘Edit’ ruft die Deklaration der selektierten Ebene im Editor auf.

2.6.7 Das Register ‘Constraints’

Das Register ‘Constraints’ stellt alle geladenen Constraints dar. Sie werden alphabetisch nach Bezeichnern geordnet aufgelistet. Zu jedem Constraint wird angegeben, in welcher Datei es definiert ist, welcher Gruppe und welchen Ebenen es angehört, welche seine Bewertung ist und ob es momentan aktiv ist oder nicht.

Der Button ‘Show’ druckt die selektierten Constraints in der Shell aus, als wäre dort der Befehl `constraint` eingegeben worden.

Der Button ‘Edit’ öffnet die Datei, die das Constraint definiert hat, an der entsprechenden Stelle. Da der externe Editor asynchron gestartet wird, wird nach dem Editieren das Constraint nicht automatisch neu geladen; dazu muß der Befehl Files:Reload ausgeführt werden.

Der Button ‘Use’ schaltet sie selektierten Constraints aus oder ein, so als wäre in der Shell der Befehl `useconstraint` auf jedes einzelne angewandt worden. Ein Knopfdruck kann also mehrere Constraints aus- und gleichzeitig andere anschalten.

Der Button ‘Use Group’ schaltet alle Constraints aus oder an, die einer selektierten Gruppe angehören. Dabei gilt eine Gruppe als selektiert, wenn sie mindestens ein selektiertes Constraint enthält. Der Aufruf ändert den Status jedes einzelnen Constraints, wenn also eine Gruppe von Constraints zuvor nur teilweise aktiv war, ist sie danach wiederum nur teilweise aktiv.

Der Button ‘Use Level’ schaltet alle selektierten Ebenen an oder aus. Dabei gilt eine Ebene als selektiert, wenn ihr mindestens ein selektiertes Constraint angehört. Die Wirkung ist, als wäre in der Shell jeweils genau einmal der Befehl `uselevel` angewandt worden. (Der Eintrag “active” der selektierten Constraints ändert sich dadurch nicht; die Veränderung ist nur im Register ‘Levels’ zu beobachten.)

2.6.8 Das Register 'Lexikon'

Das Register 'Lexikon' stellt alle bekannten Lexikoneinträge dar. Zu jedem Eintrag ist die definierende Datei, die Lautung und die Zahl der Lesarten angegeben.

Der Button 'Display' zeigt die vollständige Definition der selektierten Lexikoneinträge in der Shell an, so als wäre der Befehl `lexicon` aufgerufen worden.

Der Button 'Edit' ruft den Editor mit der Definition der selektierten Einträge auf.

2.6.9 Das Register 'Hierarchies'

Das Register 'Hierarchies' zeigt alle geladenen Hierarchien an. Jede Hierarchie ist in einem eigenen Fenster dargestellt, zwischen denen durch Registerzungen gewählt werden kann. Zur besseren Darstellung kann die Abfolge der Kinder eines Knotens von der Reihenfolge bei der Definition abweichen.

Wenn möglich werden Hierarchien als Bäume dargestellt. Bei nicht baumartigen Graphen werden die Knoten gruppiert nach maximaler Entfernung vom Wurzelknoten, die Väter jeweils linkerhand, die Söhne rechterhand. Bei Bewegen des Mauszeigers auf einen Vaterknoten werden dessen Söhne farbig unterlegt. Es wird empfohlen, zur Betrachtung einer umfangreichen Hierarchie die Shell auszublenden (Menü Window) und das Xcdg-Fenster zu vergrößern.

2.7 Die `cdg`-Shell

Das Shell-Fenster zeigt Ausgaben des Parsers, Warnungen, Debuginformation und Fehlermeldungen an und nimmt Eingaben des Benutzers an. Die Befehlssyntax ist dem originalen Programm `cdg` nachempfunden, aber sie ist keine Schnittstelle zu diesem Programm selbst; vielmehr sind alle wichtigen Funktionen re-implementiert. Es bestehen eine Reihe von Unterschieden zum Verhalten der nichtgraphischen Version:

- Argumente, die Leerstellen enthalten (wie etwa der typische Wert der Variablen 'ic-params') müssen in doppelte Anführungszeichen eingefaßt werden, nicht wie under CDG in einfache Anführungszeichen.
- Informationen, Warnungen, Fehler und Debugausgaben werden farblich hervorgehoben.
- Die Shell zeigt keine Eingabeaufforderung an, nachdem ein Befehl terminiert ist, sondern erst nach der Eingabe eines Zeilenvorschubs.
- automatische Vervollständigung mit Hilfe von TAB funktioniert nur für Befehlsnamen, nicht für die Namen von Argumenten.
- Fehlermeldungen können von denen der Originalversion abweichen.
- Der Befehl `help` existiert nicht.

Einige zusätzliche Befehle sind gegenüber dem Programm CDG verfügbar:

Befehl	Wirkung
`clear`	leert das Eingabefenster
`cd`	wechselt das Arbeitsverzeichnis
`deleteparse`	wie 'Parses::Delete'
`get`	zeigt einzelne CDG-Variable an
`printf`	wie die C-Funktion
`puts`	gibt das Argument unverändert aus
`pwd`	zeigt das aktuelle Arbeitsverzeichnis an
`run`	führt die angegebene Datei als XCDG-Skript aus
`showparse`	wie 'Parses::Tree'

Außerdem liefert jeder Befehl einen Rückgabewert, der zur Weiterverarbeitung genutzt werden kann. So ist etwa diese Eingabe legal:

```
cdg> net [newnet T0]
```

und bewirkt dieselbe Verarbeitung wie die Eingabe

```
cdg> newnet T0
cdg> net T0
```

2.8 Beispielsitzung

Eine einfache Sitzung mit `Xcdg` kann etwa so aussehen:

- `Xcdg` aus der Shell aufrufen: `$ xcdg &`
- die Arbeitsgrammatik laden, also z.B. ein Skript ausführen, das alle verwendeten Dateien lädt: File:Run(grammar.scr)
- einen Dependenzbaum erzeugen: Wordgraphs:New Net(Satz1)
- das Netzwerk absuchen: Networks:Search(net0)
- die Lösung überprüfen: Parses:Show(net0#1)
- Programm beenden: File:Quit

Bibliography

Foth, Kilian. 1999. Transformationsbasiertes Constraint-Parsing. Diplomarbeit, Fachbereich Informatik, Universität Hamburg.

Glover, F. 1989. Tabu search - part I. *ORSA Journal on Computing*, 1(3).

Glover, F. 1990. Tabu search - part II. *ORSA Journal on Computing*, 2(1).

Harper, Mary P. and Randall A. Helzerman. 1994. Managing multiple knowledge sources in constraint-based parsing of spoken language. Technical Report EE 94-16, School of Electrical Engineering, Purdue University, West Lafayette, IN.

Harper, Mary P., L. H. Jamieson, C. D. Mitchell, G. Ying, S. Potisuk, P. N. Srinivasan, R. Chen, C. B. Zoltowski, L. L. McPheters, B. Pellom and R. A. Helzerman. 1994. Integrating language models with speech recognition. In *Proceedings of the AAAI-94 Workshop on the Integration of Natural Language and Speech Processing*, pages 139–146.

Harper, Mary P., Leah H. Jamieson, Carla B. Zoltowski and Randall A. Helzerman. 1993. Semantics and constraint parsing of word graphs. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, pages 63–66, Minneapolis, MN.

Heinecke, Johannes, Jürgen Kunze, Wolfgang Menzel and Ingo Schröder. 1998. Eliminative parsing with graded constraints. In *Proceedings of the Joint Conference COLING/ACL-98*, Montréal, Canada.

Maruyama, Hiroshi. 1990a. Constraint dependency grammar. Technical Report RT0044, IBM Research, Tokyo Research Laboratory.

Maruyama, Hiroshi. 1990b. Structural disambiguation with constraint propagation. In *Proceedings of the 28th Annual Meeting of the Association of Computational Linguistics (ACL-90)*, pages 31–38, Pittsburgh, PA.

Maruyama, Hiroshi, Hideo Watanabe and Shiho Ogino. 1990. An interactive Japanese parser for machine translation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 257–262, Helsinki.

Menzel, Wolfgang. 1994. Parsing of spoken language under time constraints. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 560–564, Amsterdam.

Menzel, Wolfgang. 1995. Robust processing of natural language. In *Proceedings of the 19th German Annual Conference on Artificial Intelligence (KI-95)*, pages 19–34, Berlin.

Schröder, Ingo. 1995. Analyse natürlicher Sprache durch Beschränkungserfüllung. Studienarbeit, Fachbereich Informatik, Universität Hamburg.

Schröder, Ingo. 1996. Integration statistischer Methoden in eliminative Verfahren zur Analyse von natürlicher Sprache. Diplomarbeit, Fachbereich Informatik, Universität Hamburg.

Schröder, Ingo. 1997a. Benutzerhandbuch des CDG-Parsers 0.2. Memo HH-2/97, Projekt DAWAI, Fachbereich Informatik, Universität Hamburg.

Schröder, Ingo. 1997b. Benutzerhandbuch des CDG-Parsers 0.8. Memo HH-4/97, Projekt DAWAI, Fachbereich Informatik, Universität Hamburg.

Schröder, Ingo. 1997c. Syntax der Eingaben in den CDG-Parser 0.2. Memo HH-1/97, Projekt DAWAI, Fachbereich Informatik, Universität Hamburg.

Schröder, Ingo. 1997d. Syntax der Eingaben in den CDG-Parser 0.8. Memo HH-3/97, Projekt DAWAI, Fachbereich Informatik, Universität Hamburg.

Schulz, Michael. 2000. Parsen natürlicher sprache mit gesteuerter lokaler Suche. Diplomarbeit (in Vorbereitung), Fachbereich Informatik, Universität Hamburg.

Voudouris, Christos. 1997. *Guided Local Search for Combinatorial Optimisation Problems*. Ph.D. thesis, Department of Computer Science, University of Essex, Colchester, UK.