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

1 The cdg parser **4**

1.1 Command line format 4

1.2 Interactive commands 5

1.2.1 The command 'activate' 5

1.2.2 The command 'anno2parse' 7

1.2.3 The command 'annos2prolog' 7

1.2.4 The command 'annotation' 7

1.2.5 The command 'chunk' 8

1.2.6 The command 'compareparses' 8

1.2.7 The command 'compile' 8

1.2.8 The command 'constraint' 8

1.2.9 The command 'dbclose' 8

1.2.10 The command 'deactivate' 9

1.2.11 The command 'distance' 9

1.2.12 The command 'edges' 9

1.2.13 The command 'frobbing' 9

1.2.14 The command 'gls' 10

1.2.15 The command 'help' 16

1.2.16 The command 'hierarchy' 17

1.2.17 The command 'hook' 17

1.2.18 The command 'incrementalcompletion' 18

1.2.19 The command 'inputwordgraph' 18

1.2.20 The command 'isearch' 18

1.2.21 The command 'level' 18

1.2.22 The command 'levelsort' . . . 19
1.2.23 The command 'lexicon' . . . 19
1.2.24 The command 'license' . . . 19
1.2.25 The command 'load' . . . 20
1.2.26 The command 'ls' . . . 20
1.2.27 The command 'net' . . . 20
1.2.28 The command 'netdelete' . . . 21
1.2.29 The command 'netsearch' . . . 22
1.2.30 The command 'newnet' . . . 24
1.2.31 The command 'nonspeccompatible' . . . 24
1.2.32 The command 'parsedelete' . . . 24
1.2.33 The command 'parses2prolog' . . . 24
1.2.34 The command 'predict' . . . 25
1.2.35 The command 'printparse' . . . 27
1.2.36 The command 'printparses' . . . 27
1.2.37 The command 'quit' . . . 27
1.2.38 The command 'shift-reduce' . . . 27
1.2.39 The command 'renewnet' . . . 28
1.2.40 The command 'reset' . . . 28
1.2.41 The command 'section' . . . 28
1.2.42 The command 'set' . . . 29
1.2.43 The command 'showlevel' . . . 34
1.2.44 The command 'status' . . . 34
1.2.45 The command 'tagger' . . . 35
1.2.46 The command 'testing' . . . 35
1.2.47 The command 'useconstraint' . . . 35
1.2.48 The command 'uselevel' . . . 35
1.2.49 The command 'uselexicon' . . . 36
1.2.50 The command 'verify' . . . 36
1.2.51 The command 'version' . . . 36
1.2.52 The command 'weight' . . . 36
1.2.53 The command 'wordgraph' . . . 37
1.2.54 The command 'writeannotation' . . . 37

1.2.55 The command 'writenet' . . . 38
1.2.56 The command 'writeparses' . . . 38
1.2.57 The command 'writewordgraph' . . . 38
1.3 Example session . . . 38
1.4 Grammar elements . . . 41
1.4.1 Levels of analysis . . . 41
1.4.2 Constraints . . . 42
1.4.3 Functions . . . 47
1.4.4 Predicates . . . 49
1.4.5 Lexicon entries . . . 52
1.4.6 Hierarchies . . . 54
1.4.7 Data maps . . . 55
1.4.8 Word graphs . . . 55
1.4.9 Annotations . . . 55

2 The Visualisator `xcdg` 59
2.1 Introduction . . . 59
2.2 Invocation . . . 59
2.3 The `xcdg` window . . . 60
2.4 The menu bar . . . 60
2.5 The CDG shell . . . 60
2.5.1 Additional Commands . . . 62
2.6 The Data Browser . . . 63
2.6.1 The Files Browser . . . 64
2.6.2 The Lattice Browser . . . 64
2.6.3 The Constraint Net Browser . . . 65
2.6.4 The Parse Browser . . . 65
2.6.5 The Levels Browser . . . 65
2.6.6 The Constraints Browser . . . 66
2.6.7 The Lexicon Browser . . . 66
2.6.8 The Hierarchy Browser . . . 66
2.7 The Tree Editor . . . 67
2.7.1 The Tree Editor Menu . . . 67
2.7.2 The Tool Bar . . . 69
2.7.3 The Conflict List . . . 70
2.7.4 The Tree Window . . . 70
2.8 User and Expert Mode . . . 74

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
```

Category	Command	Description
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
`greedy <yes|no>`	always take global improvements?	yes
`method <name>`	frobbing method	`combined`
`maxsize <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.

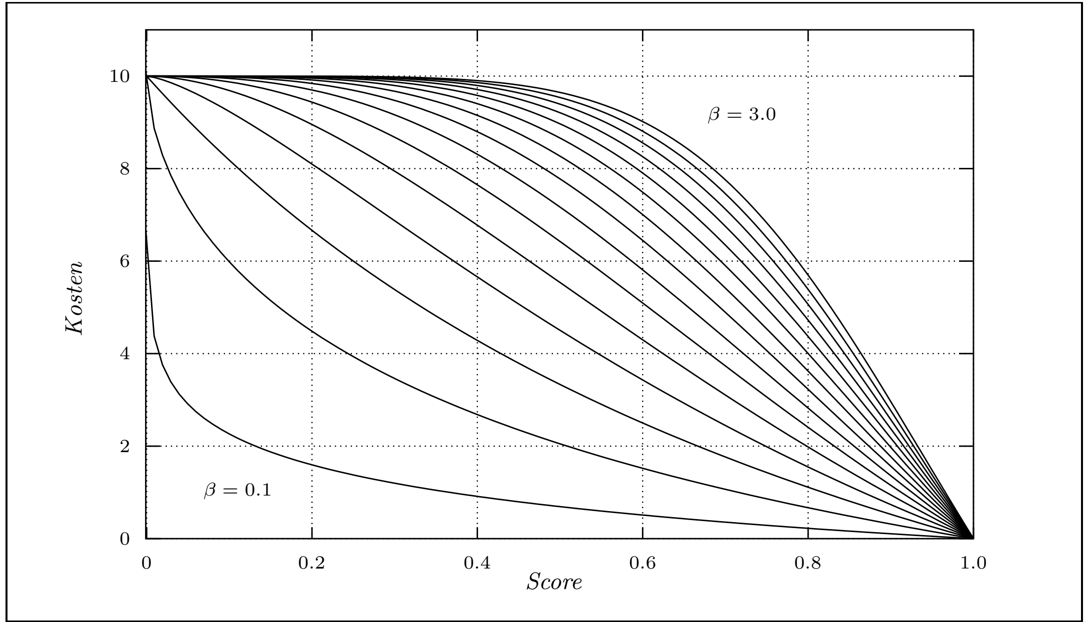

Figure 1.1: Conversion of scores into costs

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 1.2.42). 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>
```

Note that you need to separate punctuation marks from normal words with whitespace in order to have them recognized as separate tokens. Alternatively, you can set the variable `tokenizer` to specify a more intelligent program that tokenizes the line properly.

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]
```
$\Longrightarrow$ `6 krankenhaus{3}(1-2)/SEM 4:` $\Longleftarrow$
```
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]
```
$\Longrightarrow$ `26 liegt{2}(2-3)/SEM 1:  ROOT-NIL[1]` $\Longleftarrow$
```
[...]
```

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 'predict'

This command adds an external program to be called whenever a parsing problem is created.

The command takes two arguments: the name of a knowledge source and the complete invocation that will be executed. The name should be a descriptive string, but is not otherwise used by CDG. The invocation need not contain the full path to the program if it can be found in the PATH environment variable.

Whenever a new constraint net is created, the specified program will be run on the list of words (and syntactical categories if `taggerCategoryPath` is set) of the word graph in question. Its output will be scanned for predictions that are then available for use in constraints via the 'predict' function.

Example: a part-of-speech tagger could be registered by the command

```
predict tagger my-tagger.sh
```

When a new constraint net is created from the sentence 'Sag mir die Wahrheit!', this command will be executed on a file with the contents

```
sage
mir
die
Wahrheit
!
```

and will (hopefully) reply with

```
sage    VVIMP   1.000
mir     PPER    1.000
die     ART     1.000
Wahrheit        NN      1.000
!       $.      1.000
```

After this, the formula `predict(X@id, tagger, VVIMP)` will evaluate to 1 when applied to the first word in the lexeme graph, while the formula `predict(X@id, tagger, VVFIN)` will evaluate to 0.

A predictor can request more information from the program than just the sequence of words. Any additional arguments to the '`predict`' command are interpreted as feature requests. For instance, a predictor could be interested in the base form of each word, and the category that POS tagging predicted for it. Such a predictor can be registered with the command

```
predict attacher attach.pl cat base
```

and might then receive the input

```
sage       VVIMP        sagen
mir        PPER         -
die        ART          -
Wahrheit   NN           Wahrheit
!          $!           -
```

`cdg` computes the strings that are passed to the predictor in the following way:

- If the request is 'cat', it searches the predictions that were already made for each word. If a prediction is found whose value is a number, the prediction whose value is the highest number is returned. This can be used in combination with previous POS tagging to determine the probable category of a word.
- All other strings are interpreted as attributes of the CDG lexicon. All lexicon items that are available for the word are checked; if one is found that has the requested attribute, its value for that attribute is printed. If no such lexicon item can be found, the string '-' is printed.
- If the 'cat' request was given, lexicon items whose value for the attribute `taggerCategoryPath` matches the preferred category are preferred when deciding between different possible values of the current request.

  Example: Assume that POS tagging has been performed, and a predictor requests the base form ('base') of the word 'gefällt'. Assume further that the lexicon contains different items with that reading, a finite form derived from 'gefallen' and a participle derived from 'fällen'. If POS tagging has established that the word is a finite verb, the correct value 'gefallen' can be printed rather than the less probable 'fällen'.

The mechanism can be used to attach arbitrary information to a sentence, if it is available. A hypothetical perfect analyser could be registered with

```
predict oracle magic.sh
```

and might reply with

```
sage       label S      parent 0
mir        label OBJD   parent 1
die        label DET    parent 4
Wahrheit   label OBJA   parent 1
!          label ROOT   parent 0
```

Such an oracle can be approximated e.g. by supertagging.

If several predictors are registered via 'predict', they will be called in the order in which they were registered.

To remove a predictor again, specify 'off' as the second parameter. (You cannot use a script called 'off' to generate predictions, sorry.)

1.2.35 The command 'printparse'

This command displays the specified dependency analysis in textual form. Syntax:

```
printparse [-f log|xml|cda] [<parse-id>]
```

Without any arguments it prints the most recently created parse, i.e. created using a solver (netsearch, frobbing, gls ...). The `<parse-id>` identifies a specific parse (like `parse0`, `parse1` ...) that is to be shown. The option `-f` allows to choose between three output formats:

- log: human readable output format (default)
- xml: print the parse using xml markup
- cda: format to reuse parses as annotations

Note, that the `printparse` command will output xml markup when the global `xml` variable is set (see 1.2.42) even though `-f xml` is not used.

1.2.36 The command 'printparses'

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

1.2.37 The command 'quit'

The command 'quit' terminates the program.

```
cdgp> quit
bye
%
```

1.2.38 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.

1.2.39 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
```

1.2.40 The command ‘reset’

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

1.2.41 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
```

1.2.42 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 variable specifies directories 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`. More than one directory can be given, separated with commas; they will be searched in the specified order, and if the search in one directory is successful, later ones will not be searched.

  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`.

- '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.

- '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'.

- 'timelimit': Indicates the number of milliseconds after which the search for a dependency analysis is aborted. The value zero means unlimited.

- 'tokenizer': If this variable is set to a string, arguments to the Command 'inputwordgraph' are piped through a call to that program before being translated into arcs in the lexeme graph. The program must be a filter (read from STDIN and print to STDOUT) and print one tokenized word per line.

For instance, in an utterance such as "Stop!", cdg would naïvely assume that it contains only one word because there is no whitespace in it. A proper tokenizer would recognize four words instead. This makes it easier to cut and paste normal text into 'inputwordgraph'.

- '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
```

1.2.43 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
```

1.2.44 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
```

1.2.45 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.

1.2.46 The command 'testing'

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

1.2.47 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
```

1.2.48 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
```

1.2.49 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.

1.2.50 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?

1.2.51 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.
```

1.2.52 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. If the first argument is not the name os a constraint, it may also be the name of a constraint section; in this case all constraints from that section change their weights.

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>
```

1.2.53 The command 'wordgraph'

The command 'wordgraph' lists some or all loaded word graphs. If no parameters are given, all loaded word graphs are listed (but not those that would be autoloaded if they were named explicitly).

```
cdgp> wordgraph
heiseticker-s1 : Konkursgerüchte, drücken, Kurs, der, Amazon-Aktie;
heiseticker-s2 : Begleitet, von, Marktgerüchten, über, den, bevorstehenden, Konkurs, von, Amazon, ,,
heiseticker-s3 : An, der, Nasdaq, rutschte, das, Papier, am, gestrigen, Mittwoch, kurz, sogar, unter
...
```

If parameters are given, only those word graphs are shown whose label was explicitly given.

```
cdgp> wordgraph heiseticker-s1
heiseticker-s1 : Konkursgerüchte, drücken, Kurs, der, Amazon-Aktie;
cdgp>
```

If a parameter is not the name of a word graph, word graphs whose name includes the parameter are shown instead.

```
cdgp> wordgraph s1
heiseticker-s1 : Konkursgerüchte, drücken, Kurs, der, Amazon-Aktie;
cdgp>
```

If the first parameter is '-c', then all word graphs which *contain* the second parameter as a word are shown.

```
cdgp> wordgraph -c Apple
heiseticker-s2908 : Quicktime-Hersteller, Apple, hatte, nach, Erscheinen, der, ersten, Beta-Version,
heiseticker-s2914 : Doch, viele, Hersteller, wie, Apple, ,, Flash, und, Adobe, haben, ihre, Software
heiseticker-s3209 : Apple, iBook, :;
...
```

1.2.54 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.

1.2.55 The command **'writenet'**

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

1.2.56 The command **'writeparses'**

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

1.2.57 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 (\).

1.4.1 Levels of analysis

Level declarations are written like this:

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

[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)

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.

BUG: Explain what 'mirror' levels are and how to specify them.

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);
```

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).

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)

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

```
// JUNK immer unter ROOT            // JUNK immer unter ROOT
{X:SYN} : syn_junk : 0.0 :          {X!SYN} : syn_junk : 0.0 :
```

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)

```
~root(X^id) -> X.label!=JUNK;      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:

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

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:

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

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. Note that `=` and `!=` are *not* exactly contrary: if at least one of their operands is undefined (e.g. an access to a feature that the lexicon item does not contain, or any access to the regent of a NIL edge), *both* operators always return `false`. This means that the two formulas `X@foo != bar` and `~(X@foo = bar)` are not completely equivalent: if `X@foo` is not defined, the first will return `false`, but the second will return `true`. In other words, the undefined value is neither equal nor unequal to anything, not even itself.

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
- 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.

[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.

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 timepoint of 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@to;
```

  Obviously this is only useful to find the parent of the *regent* of a particular edge `X`; the parent of its *dependent* is already available via the `^` operator, so instead of `parent(X@id)`, always write simply `X^id`.

  The name of a unary constraint can be given as a second argument; the function will then ascend the syntax tree recursively until it encounters an edge that does not satisfay this constraint. If no unary constraint of that name exists, the second argument is taken to be a label of the level that the edge belongs to; the search then continues as long as intermediate edges carry this label.

- '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.)
- 'predict': This function lets you access predictions that were made by external components on the basis of the sequence of words in the input. The first parameter is the word to which the requested prediction was attached; this is typically a term such as `X@id`. The second parameter is the name of the predictor whose output should be read, and the third parameter is the requested key. The return value is the value that the predictor predicteed for that key. Depending on what property was previously assigned, the result may be a number or a string.

  A common use of this functionality is via the integrated POS tagger interface. If a POS tagger was active when the lexeme graph was built, then each word will carry

a prediction about how likely the tagger said its category was. The keys into this table are the syntactic categories used by the tagger, and the values are the respective probabilities that it predicted. For instance, a token with the reading 'der' might carry a prediction of 1.0 for the key 'ART' and of 0.1 for the key 'PRELS'.

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, predict(X@id, POS, X@cat)) ] :
  predict(X@id, POS, X@cat) = 1.0;
```

Another predefined key for each word is the syntactic category that it *really* has. This can only be determined when the `taggerCategoryPath` variable is set and an annotation can be found for the lattice in question that specifies this attribute. For instance, the same token 'der' in the example above might carry a property 'cat' with the value 'ART'. Reading this property is of course only allowed to measure the effect that perfect part-of-speech tagging would have on a a grammar.

- '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.

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.

[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.

- '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);
```

  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.

  As with `has`, the second argument may also be the name of a unary constraint rather than an edge label; in this case, the constraint is called on the tested edge, and the `is` returns thhe result of this invocation.

  A third and fourth parameter can also be specified, just like for `has`; the third specifies a hierarchy within which the second parameter will be used as an inner node for 'subsumes' checks instead of equality tests. The fourth specifies another inner node in the same hierarchy whose descendants are to be skipped recursively. For instance, if coordinations are chained with the label 'COORD', then case agreement of structurally distant conjuncts can be checked by saying

```
{X!SYN} : 'Subjekt-Kasus' :
X.label = COORD &
is(X^id, SUBJ, Label, COORD)
->
X@case = nom;
```

  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 often 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.

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

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,
  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)

<table>
<tr><td>AnnoId</td><td>::=</td><td>String</td></tr>
<tr><td>Arc</td><td>::=</td><td>Stars From To Word [Penalty]</td></tr>
<tr><td></td><td>|</td><td>Stars Word</td></tr>
<tr><td>ArcList</td><td>::=</td><td>ArcList ‘,’ Arc</td></tr>
<tr><td></td><td>|</td><td>Arc</td></tr>
<tr><td>From</td><td>::=</td><td>Number</td></tr>
<tr><td>Penalty</td><td>::=</td><td>Number</td></tr>
<tr><td>Stars</td><td>::=</td><td>‘*’ [Stars]</td></tr>
<tr><td>To</td><td>::=</td><td>Number</td></tr>
<tr><td>Word</td><td>::=</td><td>String</td></tr>
<tr><td>WordGraph</td><td>::=</td><td>WordGraphId ‘:’ AnnoId ‘:’ ArcList</td></tr>
<tr><td>WordGraphId</td><td>::=</td><td>String</td></tr>
</table>

Table 1.6: Word graphs (EBNF)

<table>
<tr><td>AnnoId</td><td>::=</td><td>String</td></tr>
<tr><td>Annotation</td><td>::=</td><td>From To Word SpecSeq</td></tr>
<tr><td>AnnoEntry</td><td>::=</td><td>AnnoId ‘¡-¿’ AnnoList</td></tr>
<tr><td>AnnoList</td><td>::=</td><td>AnnoList ‘,’ Annotation</td></tr>
<tr><td></td><td>|</td><td>Annotation</td></tr>
<tr><td>DepKind</td><td>::=</td><td>String</td></tr>
<tr><td>From</td><td>::=</td><td>Number</td></tr>
<tr><td>Label</td><td>::=</td><td>String</td></tr>
<tr><td>Position</td><td>::=</td><td>Number</td></tr>
<tr><td>Specification</td><td>::=</td><td>TagKind ‘/’ TagName</td></tr>
<tr><td></td><td>|</td><td>DepKind ‘->’ Label ‘->’ Position</td></tr>
<tr><td>SpecSeq</td><td>::=</td><td>empty</td></tr>
<tr><td></td><td>|</td><td>SpecSeq Specification</td></tr>
<tr><td>TagKind</td><td>::=</td><td>String</td></tr>
<tr><td>TagName</td><td>::=</td><td>String</td></tr>
<tr><td>To</td><td>::=</td><td>Number</td></tr>
<tr><td>Word</td><td>::=</td><td>String</td></tr>
</table>

Table 1.7: Annotations (EBNF)

<table>
<tr><td>Hierarchy</td><td>::=</td><td>Id ‘– >’ Sort</td></tr>
<tr><td></td><td>|</td><td>Id ‘– >’ SubsumptionList</td></tr>
<tr><td>Sort</td><td>::=</td><td>String ‘(’ SortList ‘)’</td></tr>
<tr><td></td><td>|</td><td>String</td></tr>
<tr><td>SortList</td><td>::=</td><td>Sort</td></tr>
<tr><td></td><td>|</td><td>SortList ‘,’ Sort</td></tr>
<tr><td>SubsumptionList</td><td>::=</td><td>SubsumptionList ‘,’ Subsumption</td></tr>
<tr><td></td><td>|</td><td>Subsumption</td></tr>
<tr><td>Subsumption</td><td>::=</td><td>String ‘– >’ TypeSeq</td></tr>
<tr><td></td><td>|</td><td>String ‘< –’ TypeSeq</td></tr>
<tr><td>TypeSeq</td><td>::=</td><td>TypeSeq String</td></tr>
<tr><td></td><td>|</td><td>String</td></tr>
</table>

Table 1.8: Hierarchy definitions (EBNF)

Chapter 2

The Visualisator xcdg

2.1 Introduction

xcdg is a graphical shell around the cdg functionality. In principle, it duplicates the entire command set of the command-line parser cdgp and adds methods for displaying and editing syntax trees graphically. The main new features are

- visualization of defined hierarchies and dependency trees
- editing of dependency trees
- interactive grammar evaluation (changes to a dependency tree are immediately resubmitted to the active grammar)

2.2 Invocation

To call the graphical parser xcdg, specify the '-x' option to the program cdg. Alternatively, the graphical parser is called if the program is called under the name 'xcdg' (e.g. by linking it to that name).

X11 resources for the program are read from a file called Cdgrc. For instance, these resources specify the color scheme of dependency trees:

```
*ParseTree.nodeColor:           gray50
*ParseTree.lineColor:           black
*ParseTree.vlineColor:          gray
*ParseTree.fontFamily:          Helvetica
*ParseTree.fontSizes:           -7 -9 -10 -12 -14 -16 -18
*ParseTree.wordColor:           black
*ParseTree.labelColor:          black
*ParseTree.highlightColor:      white
*ParseTree.errorColor:          red
```

Usually the program uses the version of Cdgrc in the directory where `xcdg` is installed. To use different resources, create your own version of that file and load it in your `.xcdgrc` via the 'options' command (see 2.5.1).

Upon startup, `xcdg` loads the file `.xcdgrc` and runs all commands in it. This file is searched first in the current directory and then in the home directory of the user. `xcdg` does *not* run the user's .cdgrc file by default. To have this happen, you can the 'run' command in the .xcdgrc file (see 2.5.1).

The following command-line options are unique to `xcdg`:

- `-grammarpath <path>` sets a path to load grammars from. All this does is to set the default value in file selection dialogs.
- `-noinit` inhibits processing of `~/.xcdgrc`.
- `-user` specifies starting up in 'user mode' (see 2.8).

2.3 The `xcdg` window

The `xcdg` window (see Figure 2.1) combines a data browser with the normal cdg command prompt. (Either component can be switched off via the 'Window' menu.) A menu bar gives access to some global settings, and an echo area below the shell displays context-sensitive help and progress information. The light bulb in the bottom right corner blinks when the program is engaged in a long computation. Usually such computations can be interrupted by pressing Control-C in the shell.

2.4 The menu bar

The menu bar provides access to some but not all settings of the program. The 'File' menu, as usual, deals with loading input and operating the program itself. The 'Settings' menu contains tick boxes that duplicate the 'set' command. The 'Window' menu allows display of only the data browser or the shell.

BUG: setting variables via 'set' in the shell does not always propagate properly to the status of the tick boxes! In general, the menu system is a hasty shell around functionality already present elsewhere, and often inconsistently done. Use typed commands preferably.

2.5 The CDG shell

The shell window displays informational, warning, debug, and error messages of the program and receives typed input from the user. The shell window operates in almost but not quite the same way as the command-line version of `cdg`. Here is a list of differences:

- `cdgp` uses the `readline` library for command editing, while `xcdg` uses a custom implementation built around Tcl strings. This means that details of the keybindings differ slightly:

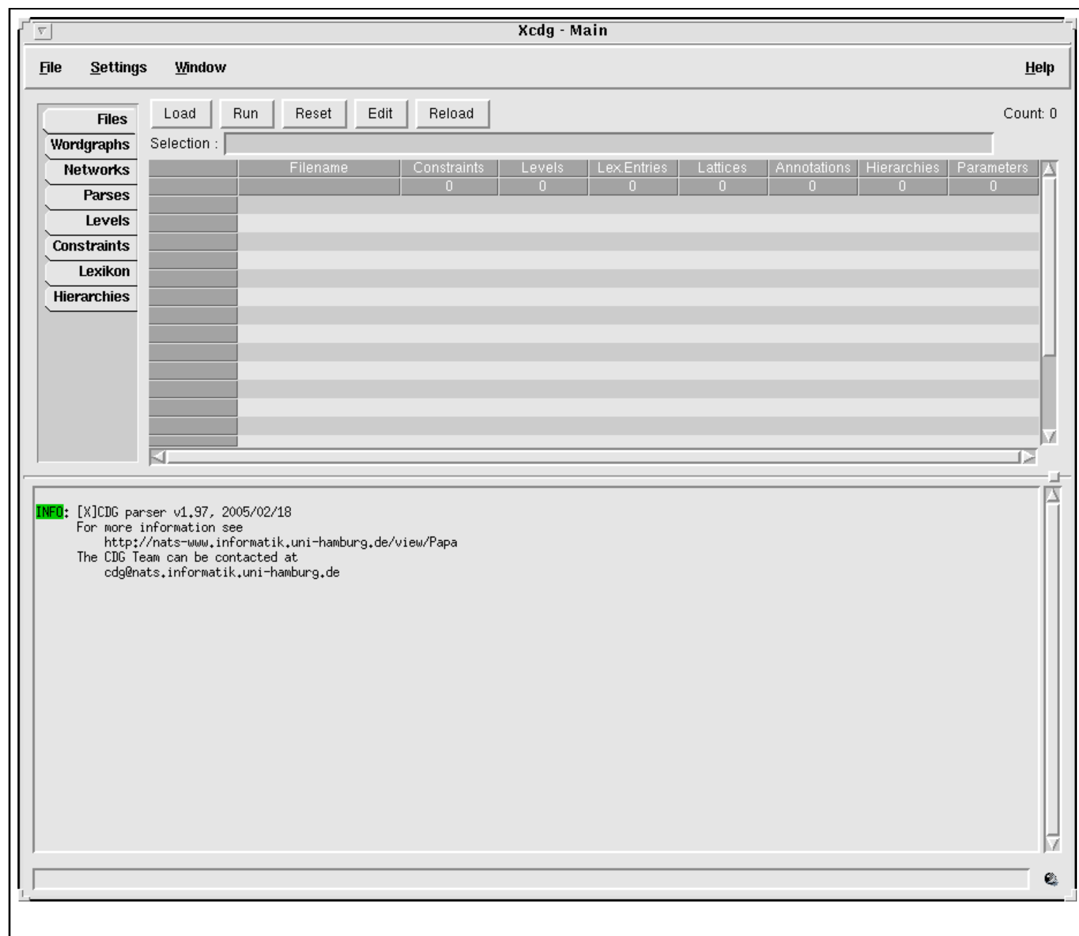

Figure 2.1: The `xcdg` window

  - The TAB key does not always generate exactly the same set of possible completions
  - typing ahead during a computation is not possible
  - many advanced features of `readline` such as 'upcase-word' are missing altogether

- Arguments that contain whitespace must be quoted with single quotes. Double quotes are *not* supported and lead to misparsed command lines.

- Also, the syntax `"a` as an input method for `ä` is not available. You must type actual umlauts (with a nationalized keyboard, or with the Compose key).

- The type designators 'INFO', 'WARNING' etc. are highlighted in various colors to indicate the status of messages.

- The shell sometimes does not display a prompt when it would be expected, for instance immediately after starting up.

- The command '`help`' does not exist.

- Commands have return values that can be used as the arguments to another command. Together with Tcl's square brackets for command evaluation this allows nested commands such as `net [newnet T0]` with the same meaning as `newnet T0` followed by `net`.
- Several commands can be juxtaposed with the semicolon. This commandline works in XCDG, but not in cdgp:

```
cdg> newnet ; netsearch ; verify
```

- Some additional commands are available (see below).

2.5.1 Additional Commands

In comparison to the command-line tool, there are additional commands that can be executed. Most of them are simply Tcl commands that can be evaluated in the Tcl interpreter; for instance, 'cd ..' changes the working directory to the parent directory, and 'clear' erases the output of all previous commands. Consult the Tcl documentation for full details on Tcl commands.

- The command 'deleteparse' removes a parse from the system just like the 'Delete' button in the parse browser.
- The command 'options' takes the name of a file as an argument and sets X resources as specified by that file.
- The command 'run' takes the name of a file as an argument and executes all `xcdg` commands found in that file.
- The most important new command is 'showparse'. In its most basic form, it takes the name of an existing parse and displays it graphically in a window of its own (see 2.7). If no parse of that name is known, the argument is interpreted as the name of an annotation instead, and if an annotation of that name exists it will be converted to a parse and then displayed. This works even if the annotation has to be autoloaded first; in particular, it will settle for an annotation name of which the given name is merely a suffix. In other words, if you distribute your treebank over numerical subdirectories as explained in 1.2.42, it suffices to type 'showparse s123' in order to display the annotation with the name 'WSJ-s123'.

  In addition to the parse or annotation names, a set of dependency edge numbers can be specified which should be highlighted in the display. For instance, the command 'showparse Tree1:10,20,30' would display the annotation 'Tree1' and highlight the edges number 10, 20, and 30 (counted from the first word and then from the first level). This will cause problems if your tree names contain colons; use the variable 'colonInTreeNames' to switch off this feature.
- The command 'set' works in the same way as described in 1.2.42, but there are various additional variables controlling the behaviour of `xcdg` that can be set:
  - `colonInTreeNames`: If set to 1, `xcdg` will not interpret colons in the arguments to 'showparse' specially.

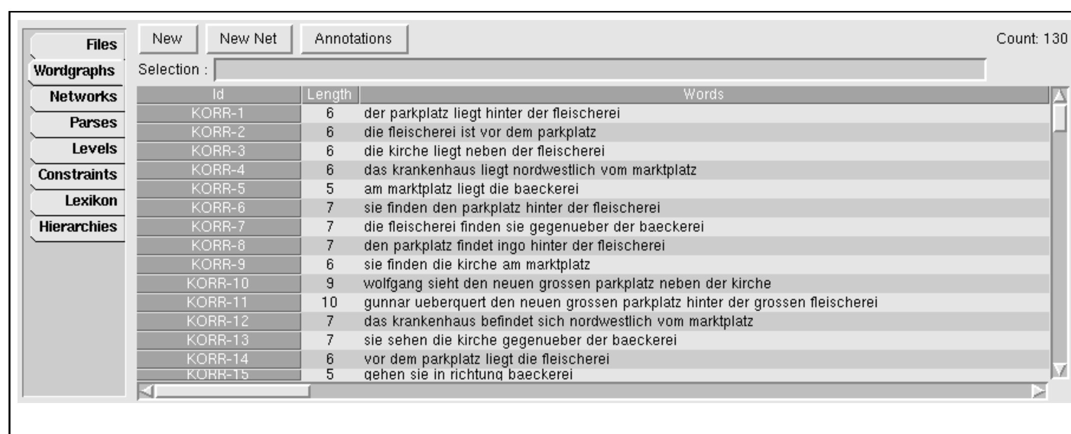

Figure 2.2: The data browser

- `confirmexit`: If set to 1 (default), `xcdg` will ask the user to confirm the '`quit`' action, whether invoked by button, command or Control-D.
- `editor`: This is the invocation of the editor that is started when the 'Edit' button is invoked in a data browser.
- `grammarpath`: sets the default path displayed in file selection dialogues.
- `manywindows`: If set to 1, parses of different sentences will open in different X windows. Otherwise, all parses are put into the same X window on different tabs of the same notebook. This is useful if you want to display all instances of a phenomenon in a tree bank, as it prevents new windows from popping up for five minutes and making your desktop unusable.
- `trace`: If set to 1, `xcdg` will display the full error messages that result from erroneous commands in the shell. Usually these messages are filtered so that only the most relevant information is printed.

2.6 The Data Browser

The data browser displays all grammar elements and computed structures, sorted by their type (see Figure 2.2).

The data browser contains

- A register for selecting data types. Left-click a tab to switch to the corresponding display.
- A series of buttons for quick command execution. Left-click a button to apply it to the current selection (see below).
- A counter indicating how many instances of a data structure are currently loaded.
- An input box for specifying a selection of arguments for buttons.

- A table of loaded data structures. All known data structures are displayed ordered alphabetically by their identifiers. Each line of the table corresponds to on data structure.

At any time, the selection contains the names of the selected lines of the table. Pressing a button applies it to all items in the selection. This works even if the command-line equivalent of that command does not allow multiple arguments. For instance, highlighting three lattices and pressing the 'New Net' button creates three new constraint nets.

Left clicking into the table highlights a row and puts the name of the corresponding data structure into the selection. By simultaneously pressing the Control key, more than one row can be highlighted. By simultaneously pressing the Shift key, all rows between the highlighted and the clicked row are highlighted. Finally, clicking the top left cell to the table selects all available items.

Furthermore, by typing into the table the first item is selected whose name starts with the typed string. For visual feedback, the typed string is displayed in the top right corner next to the counter. The TAB key may be used to complete the typed string if the next possible characters are uniquely defined. The ESC key cancels this mode of selection.

Finally, the selection may also be edited like normal text; if the highlighting of the table and the selection disagree, the selection overrides the highlight state.

2.6.1 The Files Browser

This browser displays all files loaded with the 'load' command. It also shows how many constraints, levels, etc. are loaded altogether. The files are represented by their file name alone; the full path name is shown in the echo area when the mouse is moved over a row.

The buttons 'Load' and 'Run' load or run a file like the commands of the same name. Both buttons do *not* work on the selection, since there is no point in loading an already loaded file, or treating a CDG input file as a command list. Instead they open a file selection dialog and operate on its result.

The 'Reset' button simply executes the 'reset' command.

The 'Edit' button edits the selected files with an external editor (use 'set editor' to select your favorite editor).

The 'Reload' button first resets the program and then loads all files that were loaded before.

BUG: 'reset' is broken, and reloading files manually causes crashes. When it doesn't, it takes far longer than restarting `xcdg`. Never use 'reset' or 'reload'.

2.6.2 The Lattice Browser

The lattice browser displays all loaded lattices (but not lattices that could be autoloaded if needed). Each lattice specifies its name, length and sequence of words.

The 'New' button is inoperable; use 'inputwordgraph' in the shell instead.

The 'New Net' button executes the 'newnet' command. The unary pruning factor can be specified in a separate dialog.

The 'Annotations' button finds all annotations of the selected lattice and displays them via 'showparse'.

2.6.3 The Constraint Net Browser

The constraint net browser displays all computed constraint nets. Each net specifies its name and various statistics such as the number of constraint nodes, solutions etc.

The 'Delete' button removes the selected net from memory (and from the display).

The 'Search' button executes the 'netsearch' command. Further parameters can be specified in a separate dialog.

The 'Frobbing' and 'Gls' buttons execute the respective commands with no further arguments. To pass additional arguments to the procedure you must use the shell.

2.6.4 The Parse Browser

The parse browser contains all parses present in the system. Each parse specifies a multitude of statistics such as its name, time of creation, method of creation, score, etc.

Parses are typically created either when a solution method is applied to a constraint net (for instance, 'netsearch'), or from a pre-existing annotation (for instance, via 'anno2parse'). Parses computed form a constraint net are typically named `parse0`, `parse1` etc. Parses created from an annotation bear the same name as the annotation.

The 'Delete' button removes a parse from memory. If there is a tree editor currently displaying the parse, the corresponding page is removed. If a tree editor becomes empty it will be closed altogether.

BUG: This does not actually work; removing the first parse from a tree editor leaves a tab that throws an error when clicked.

The 'Tree' button displays a parse graphically in a tree editor. If no tree editor exists into which this parse would fit, a new editor is opened. If the parse is already visible, nothing changes.

BUG: Pressing the 'Tree' button twice throws a 'bad Notebook page index' error.

2.6.5 The Levels Browser

The levels browser contains all defined levels of analysis. The levels are not sorted alphabetically, but appear in the order they were defined in. Each level specifies how many unary and binary constraints concern it, what file it was defined in, etc.

The 'Display' button prints the definition of the level in the shell as the 'level' command does.

The 'Show' button toggles the visibility of the level as the 'showlevel' command does.

The 'Use' button toggles the use of the level as the 'uselevel' command does.

BUG: Both buttons do not update the table. Pressing 'Use' really does switch off a level, but the table keeps saying 'used:yes'.

The 'Edit' button opens an editor on the definition of the level.

2.6.6 The Constraints Browser

The constraints browser displays all defined constraints ordered alphabetically by their names. Each constraint specifies where it was loaded from, to which levels it applies, what score it has and whether or not it is currently active.

The 'Show' button displays the constraint in the shell as the 'constraint' command does.

The 'Edit' button opens an editor on the definition of the constraint. Note that editing a constraint does not load the new definition; it is usually faster to start xcdg again than to load a new version of a defined constraint.

The 'Weight' button lets the user specify a new weight for that constraint, as the 'weight' command does.

The 'Use' button toggles use of the constraint as the 'useconstraint' command does.

The 'Use Group' button toggles use of the section of the constraint as the 'activate' and 'deactivate' commands do.

The 'Use Level' button toggles use of the level the constraint belongs to as the 'uselevel' does.

BUG: As with the levels browser, the 'Use' buttons do not update the table.

2.6.7 The Lexicon Browser

The lexicon browser displays all loaded lexicon items. Each lexicon item specifies its reading and the files it was loaded from.

The 'Display' button displays the specified lexicon item in the shell. It does this by calling the 'lexicon' command. Note that this will also display lexicon items that are not in the browser, but are automatically generated by lexical templates. The templates themselves are not displayed in the browser.

The 'Edit' button opens an editor on the definition of a lexicon item.

2.6.8 The Hierarchy Browser

The hierarchy browser does not contain a table, selection text box etc. Instead it displays all defined hierarchies on the pages of a tab notebook. Hierarchies that are true trees are displayed as trees. Hierarchies that are merely DAGs are displayed in parts, similar to the output of the 'hierarchy' command; the left column of each grid display contains the subsuming node and the right column contains the subsumed nodes. Touching any node highlights the nodes that are connected with it (in either direction).

BUG: Non-tree hierarchies look terrible; however, no one has ever come up with a better idea for displaying them.

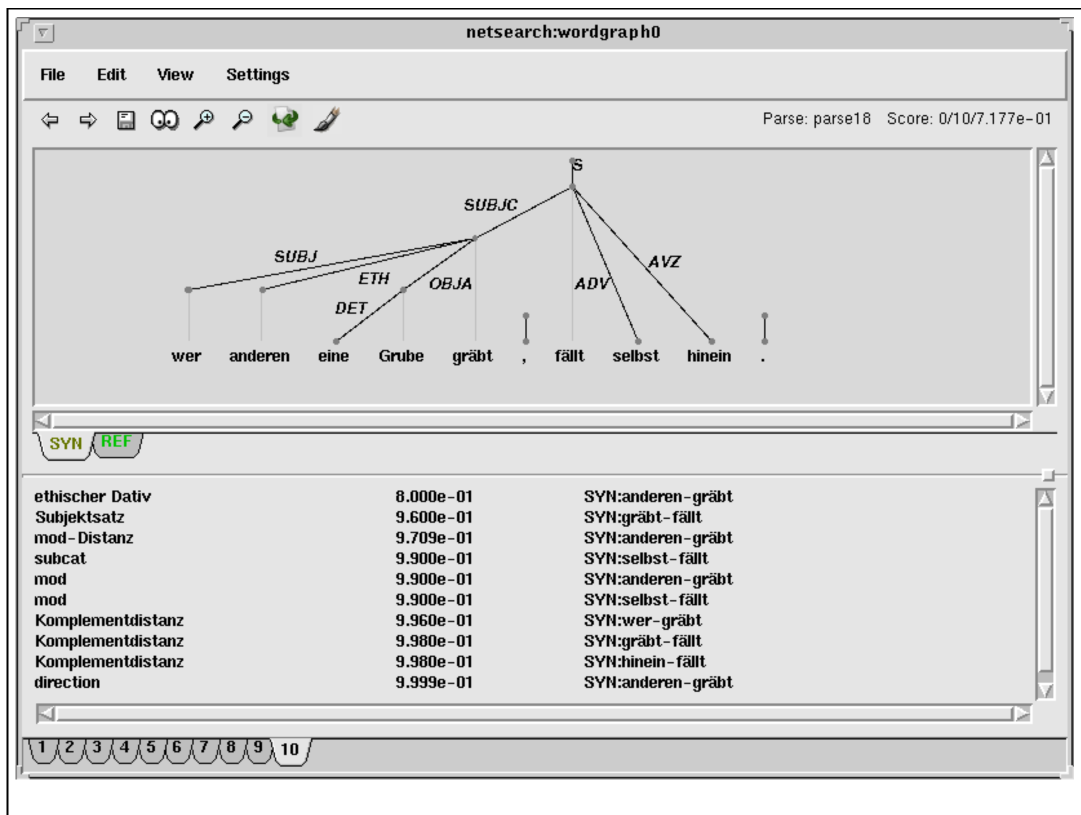

Figure 2.3: A tree editor

2.7 The Tree Editor

Parses can be displayed and edited graphically in tree editors (see Figure 2.3). A tree editor contains a menu of its own and a tab notebook of visualized parses. The tabs of the tree editor carry numbers and can be used to switch between different parses. Note that each parse has an embedded tab notebook whose tabs bear label names; these can be used to switch between different levels of the same parse.

2.7.1 The Tree Editor Menu

The menu of the tree editor contains the following items:

- File
  - Save: This writes the tree to disk as an annotation. The annotation created bears the same name as the parse. The file bears the same name plus the extension `.cda`.
  - Save as...: This saves the tree under a different name that can be specified in a file selection dialog. The extension `.cda` is added if it is not given.

- Print...: This creates a representation of the current state of the display as a postscript file. The name of the postscript file can be chosen in a file selection dialog. All sorts of highlighting are faithfully reproduced in the resulting postscript code.
- Close: This item closes the current tree editor (but not others that may be open).
- Quit: This item ends the entire `xcdg` session.

- Edit
  - Undo/Redo: These items take back or repeat changes that the user previously made to the tree.
  - Mirror: This item tries to change edges on 'mirror' levels so that they correspond to the edges of the main level (see section 1.4.1).
  - Break cycles: If the current level of the tree has cycles in it, this item changes edges in cycles to point to NIL until all cycles have been removed. It is undefined which edges are changed.
  - Parse again: This restarts the same parsing method that created the currently displayed tree. (If the tree was loaded from disk rather than parsed, frobbing is used.)

    If the user has made changes to the tree, these changes are respected; that is, if you change a label in the tree and invoke 'Parse again', XCDG will (try to) compute the best analysis that uses this particular label for that word. Of course, this may fail altogether if there is no such analysis.

- View
  - Redraw: This item draws the tree anew. This undoes the effect of clicking on constraints or highlighting edges via the extended 'showparse' command. Also, if automatic redrawing after changes to the tree structure has been switched off, this item will force a redraw.
  - Eval: This item recomputes the conflicts that the current state of the tree causes with the loaded grammar, and updates the conflict list. Note that this is usually done automatically after each change to the tree; this menu item is only necessary when you ave switched off automatic reevaluation in the Settings menu.
  - Zoom in/Zoom out: These items display the tree larger or smaller. There are seven zoom steps defined, with the middle one chosen by default.
  - Show cycles: This item highlights all edges of the current level that are part of a cycle. The 'Break cycles' item will change at least one of these edges.
  - Verify: This item calls the 'verify' command on the underlying parse and highlights all edges, labels and lexical selections that differ from the annotation of that lattice.
  - Next: This item switches to the next tree in the tree editor. If invoked on the last tree in a tree editor, it will try to find the lattice that follows the lattice of the current tree in the lattice browser, create a tree for that lattice, and display this tree in a new tab.
  - Previous: This item switches to the previous tree in the tree editor. It does nothing when invoked on the first tree in a tree editor.

- Settings
  - Auto-redraw: This toggles the behaviour of the tree after an edge has been changed. If on, the tree is automatically redrawn to look as nice as possible; this is usually what you want. However, if many changes must be made to a huge tree, the useless intermediate layout computations can take a long time, so it can be useful to switch off the behaviour, make the changes, and then re-compute the layout once. Note that even without auto-redraw the tree can be re-drawn with View::Redraw.
  - Auto-eval: This also toggles the behaviour of the tree after changes have been made. If on, the grammar is automatically reevaluated, the list of conflicts is updated, and the tree is painted anew if this feature is on. If off, the conflict list remains unchanged and will probably become out of date with respect to the tree. Even without auto-eval, an isolated evaluation can be forced with View::Eval.
  - Paint Tree: If on, each edge of the tree is automatically highlighted in the color that corresponds to the most serious constraint that it violates. This allows the user to see immediately where the trouble spots of a tree structure might be.
  - Horizontal/Vertical: These items toggle the layout of the tree itself and the list of conflicts. A horizontal layout allows more detail to be displayed about each conflict, but typically only shows the most serious conflicts. A vertical layout shows more conflicts but in a much narrower column that typically has room only for the name of each conflict; it is also better suited for displaying trees that are too high for horizontal layout.
  - Snap: These items control how an edge behaves that is dropped after a drag gesture.
    - to nearest node: the word is subordinated under the word whose node is nearest to the mouse pointer.
    - to previous mode: the subordination of the word remains the same as before.
    - to best node: the word is subordinated under the word that yields the best score for the resulting structure.

    Note that in all cases an edge that is dropped directly onto another node will snap to that node; the snap mode applies only to drop events between nodes.

2.7.2 The Tool Bar

Below the menu are some buttons for frequently required actions. They correspond to the following menu items:

- Left arrow: View::Previous
- Right arrow: View::Next
- Floppy disk: File::Save
- Xeyes: Edit::Mirror
- Lens +: View::Zoom in
- Lens −: View::Zoom out

- Recycle arrow: Settings::Auto-redraw (this button is downlighted when automatic redrawing is turned off)
- Paint brush: Settings::Paint tree
- Green arrow: Edit::Parse again

At the right end of the tool bar, the name and valuation of the underlying parse are displayed. The valuation is not the same as the score of a parse; it consists of the number of hard conflicts, the number of soft conflicts, and the product of the scores of all soft conflicts. Thus, if there are two conflicts with the scores 0.0 and 0.5, the valuation of the parse is `1/1/0.5`, while the score would be 0.0.

2.7.3 The Conflict List

The conflict list contains a record of all conflicts that a parse causes under the current grammar. Each line of the list contains the name of the failed constraint, the penalty of this conflict, and a representation of the dependency edges involved.

Note that a conflict is not the same as a constraint; it is the *instantiation* of a constraint in a particular place in a tree. For instance, a constraint that disprefers distant subordinations may well fail twice in different places in the same tree. With variable penalties, it may even have different penalties in different conflicts.

The conflict list obeys the following mouse bindings:

- A left click on a line highlights the edges involved in the conflict in a color that indicates the severity of the conflict. Penalties near 1 are symbolized by bright green, while penalties near 0 are symbolized by bright red. Intermediate penalties lead to intermediate colors between these two. The tree window is also scrolled so that the edges of the conflict are visible as well as possible.
- A middle click opens a tool tip with the definition of the violated constraint.
- A right click behaves like a left click except that for a binary conflict, the second rather than the first dependency edge is scrolled into view.

The entire conflict list is only present if a grammar is loaded that contains the same levels as the parse. It is possible to run `xcdg` on a solitary annotation and nothing else and still use 'showparse'; the resulting tree can still be edited and saved, but not evaluated.

2.7.4 The Tree Window

The tree window contains a tab notebook with one page per level of the parse. (The tabs are color-coded with the same scheme the edges of a clicked conflict.) Each level is represented as a dependency tree with the root at the top.

BUG: If a dependency tree contains cycles, the display deteriorates to an ugly bipartite graph. It should remain a normal tree and just draw the edge that causes the cycle in a special way.

The words in the sentence that is described by the tree are notated at the bottom of the tree in one line. Vertical projection lines connect each word with its corresponding inner node. Note that the tree drawing algorithm is not perfect; a non-projective tree will always contain at least one crossing between a projection line and a dependency edge, but a tree with such a crossing need not necessarily be non-projective; it might just be a tree for which `xcdg` could not figure out how to draw it without crossings. Thus, the lines are purely illustrative and have no effect whatsoever on the valuation of a parse; conditions about projectivity must be written into a constraint grammar explicitly.

If a level carries the 'mainlevel' property, then this level is treated specially: all non-NIL subordinations on other levels are drawn as additional labelled arcs below the main tree.

The tree can be edited with mouse gestures or keyboard commands. Whenever a change is made to the tree, the tree layout is recomputed for best effect (unless you switch the recomputation off under Settings::Auto-redraw). Also, the current grammar (if any) is evaluated on the resulting parse, and the conflict list is updated accordingly (unless you switch the recomputation off under Settings::Auto-eval). The following mouse gestures are available:

- Touching an element if the tree highlights it to show which element a mouse click would apply to. Additionally, touching a dependency edge also highlights its accompanying label; this can help debug the association when many long edges are very close together. Touching a word will also open a tool tip that spells out the identifier of the word. For instance, the word 'der' might bear the identifier '`der_ART`' to distinguish it as an article (as opposed to '`der_PDS`', which signifies a demonstrative pronoun).
- Clicking on a word opens a menu of homonymous lexicon items. Each line of the menu displays the identifier of a possible alternative reading. Selecting a line swaps the lexical reading of the parse.
- Middle clicking on a word opens a tool tip that displays the full lexicon definition of the current reading.
- Right clicking on a word selects the reading that leads to the best valuation of the parse in the current context. This optimization takes into account *only* the reading of the current word. If two lexical readings would have to be exchanged to reach a better valuation, for instance in a long, misinflected NP, a right click on either of them may not have the desired effect of choosing the globally optimal reading.
- Clicking on a word with the Shift key pressed opens a tool tip that lists all predictions made about the selected word. For instance, this allows you to review all predictions made by a POS tagger, and not only the best one.
- Clicking on a label opens a menu of alternative labels of the same level. Selecting one of these labels immediately exchanges the label in the tree.

  Note that it is possible to use the empty string as an edge label; such labels cannot be touched or clicked, since they never receive mouseover events. You can change them with a middle click on the corresponding edge, or via keyboard editing (see below).

  If a parse is shown but the definition of the current level has not been loaded, `xcdg` cannot determine the full set of alternative labels. Instead, it offers the user only those labels that are actually present in this tree. There is no way of assigning any other label in the tree editor; however, both the `.cda` and the `.ps` representations of trees produced by the 'Save' and 'Print' can be easily edited if this is necessary.

- Right clicking on a label optimizes it in the same way as right clicking a word. Again, only single label changes are considered; for instance, a change that would involve both a new label and a new lexical reading is never considered.
- Clicking on an edge moves the top end of the edge under the mouse pointer to indicate that the subordination is about to change. Clicking on the background canvas selects the closest edge as if it had been clicked directly.
- Middle clicking on an edge opens the label menu as if its label had been clicked on. This is useful for changing the empty label, which cannot be touched, to a non-empty label.
- Dragging an edge prepares for a subordination change. The word and the node where the edge would come to rest if dropped are highlighted at all times. Note that this does not work in 'snap to best' mode, since the necessary computation would be too slow to keep the interface responsive.
- Dropping an edge subordinates it under the place dictated by the drop policy specified under 'Settings'.

  An edge can be subordinated under NIL by dropping it onto the node at the top of another NIL edge. If there is no NIL edge in the current tree, this is not possible; in this case, use Edit::Break cycles or keyboard editing.
- Right clicking on an edge chooses the best subordination for it under the current circumstances. Again, this only considers alternative subordinations of the *current* edge, not changes to its label or word, let alone other edges.
- Right clicking on an arc optimizes it in the same way as a normal edge. Note that arcs cannot be dragged, and their labels cannot be edited at all; you must switch to the corresponding level and edit them normally there.
- Dragging the middle mouse button scrolls the canvas. Note that this gesture can be combined with normal dragging; this is necessary to move an edge between two regents that do not fit into the viewport at the same time.

  All menus opened by mouse gestures can be cancelled by pressing Escape. Also, selecting an edge and then pressing Escape returns it to the previous regent no matter what the drop policy is.

Trees can also be edited by keyboard commands alone. The following bindings are defined:

- Left/Right: These keys choose the word that the next keyboard editing command will apply to. The selected word is highlighted as if it were touched by the mouse pointer. Note that when a tree is first drawn, the first word is already selected but not highlighted. The first press of 'Right' will select the second word of the tree. Mouse editing actions also change the word selected for keyboard editing actions; this allows you to interleave mouse and keyboard editing more naturally, since both have advantages for particular tasks.
- Up: The word above the selected word is selected.
- Down: The word below the selected word is selected. If the selected word has more than one dependent, nothing happens.

- Shift: The dependency edge of the selected word is highlighted.
- Shift+Left/Shift+Right: The selected word is subordinated one word further to the left or right. A NIL binding can be created by moving the attachment point beyond the first or last word of the sentence.

  Note that the tree is not redrawn, and the grammar is not re-evaluated, until you release the Shift key. This means that moving an edge n words to one sideis not n times more expensive than moving it one word, but about the same. This behaviour is unaffected by the Auto-Redraw and Auto-Eval settings.
- Shift+Up: The selected word is subordinated to the regent of its current regent. If its regent is the root of the tree, the selected word also becomes a root.
- Shift+Down: The selected word is subordinated to a dependent of its current regent (but never to itself). If the current regent has more than one other dependent, one is chosen arbitrarily.
- Shift+Enter: The current word is subordinated to the optimal regent, as with an edge right click.
- Control: The label of the dependency edge of the selected word is highlighted.
- Control+Up/Control+Down: The label of the current word is changed to the previous or the next label in the set of possible labels for this level.
- Control+Home/Control+End: The first or last available label is chosen for the current word.
- Control+Enter: The optimal label for this subordination is selected, as with a label right click.
- Meta+Letter: This combination can be used to accelerate label selection. Letters and digits typed with the Meta key pressed are interpreted as the prefix of the new label that is selected when the Meta key is released. Thus, if your grammar contains the labels BLINKY, INKY, PINKY and SUE, pressing Meta+p is enough to choose the label PINKY. (Note that this prefix search is case insensitive.) With many labels, this can be faster than scrolling through the choices with Control+Down. You can accelerate longer prefixes by typing more than one letter before releasing the Meta key.
- Alt: The tool tip of the selected word is opened.
- Alt+Up/Alt+Down: The previous or next alternative lexical reading is chosen for the current word.
- Alt+Home/Alt+End: The first or last available lexical reading is chosen for the current word.
- Alt+Enter: The locally optimal lexical reading is chosen as with a word right click.

With all key combinations, the effect of changing the tree does not take effect until the modifier key is released; for instance, when changing the tree structure via Shift+Left the tree is not redrawn, and the grammar is not re-evaluated until the Shift key is released. This

ensures that an edge can quickly be moved by more than one word without overhead for unwanted intermediate trees.

The entire tree editor accepts the following additional key bindings:

- Control-R: Edit::Redo
- Control-Z: Edit::Undo
- Numeric −: View::Zoom out
- Numeric +: View::Zoom in
- Numeric Enter: toggle View::Horzintal/View::Vertical
- Shift+Tab: Switch to previous level
- Tab: Switch to next level
- c: View::Show cycles
- m: Edit::Mirror
- n: View::Next
- p: View::Previous
- q: File::Close
- s: File::Save
- v: View::Verify

2.8 User and Expert Mode

!WRITEME!

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.