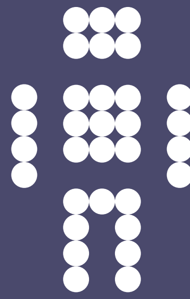
CFEngine



CFEngine Reference Manual

Auto generated, self-healing knowledge

CFEngine Core 3.5.0a1.b704ecb

CFEngine Enterprise 3.1.0a1.87abbdb

cfengine.com

Under no circumstances shall CFEngine AS be liable for errors or omissions in this document. All efforts have been made to ensure the correctness of the information contained herein.

Copyright © 2008,2010 to the year of issue CFEngine AS

Table of Contents

1 CFEngine 3.5.0a1.b704ecb – Getting started 1
1.1 Software components 1
1.1.1 cf-agent 1
1.1.2 cf-execd 2
1.1.3 cf-know* 2
1.1.4 cf-monitord 2
1.1.5 cf-promises 2
1.1.6 cf-runagent 2
1.1.7 cf-serverd 2
1.1.8 cf-report 2
1.1.9 cf-key 2
1.1.10 cf-hub 3
1.2 Core concepts 3
1.3 A renewed CFEngine 3
1.4 Installation 4
1.5 Syntax, identifiers and names 5
1.6 The work directory 5
1.7 Decisions 6
1.7.1 CFEngine hard classes 6
1.7.2 Class combination operators and precedence 7
1.7.3 Global and local classes 7
1.8 Filenames and paths 9
1.9 Upgrading from CFEngine 2 9
1.10 Testing as a non-privilieged user 10
1.11 The 'bare necessities' of a CFEngine 3 11
1.12 Familiarizing yourself 11
1.13 Remote access troubleshooting 14
1.13.1 Server connection 14
1.13.2 Key exchange 15
1.13.3 Time windows (races) 16
1.13.4 Other users than root 17
1.13.5 Encryption 17

2 A simple crash course in concepts 19
2.1 Rules are promises 19
2.2 Best practice for writing promises 20
2.3 Containers 21
2.4 When and where are promises made? 22
2.5 Types in CFEngine 3 23
2.6 Datatypes in CFEngine 3 24
2.7 Variable expansion in CFEngine 3 24
2.7.1 Scalar variable expansion 24
2.7.2 List variable substitution and expansion 24

2.7.3 Special list value cf_null 27
2.7.4 Arrays in CFEngine 3 27
2.8 Name spaces 28
2.9 Normal ordering 29
2.9.1 Agent normal ordering 29
2.9.2 Server normal ordering 30
2.9.3 Monitor normal ordering 31
2.9.4 Knowledge normal ordering 31
2.10 Loops and lists in CFEngine 3 31
2.11 Pattern matching and referencing 33
2.11.1 Runaway change warning 36
2.11.2 Commenting lines 36
2.11.3 Regular expressions in paths 37
2.11.4 Anchored vs. unanchored regular expressions 39
2.11.5 Special topics on Regular Expressions 39
2.12 Distributed discovery 40

3 How to run CFEngine 3 examples 43

4 A complete configuration 45
4.1 'promises.cf' 45
4.2 'site.cf' 47
4.3 'update.cf' 53
4.4 'failsafe.cf' 55
4.5 What should a failsafe and update file contain? 56
4.6 Recovery from errors in the configuration 56
4.7 Recovery from errors in the software 58

5 Control promises 61
5.1 common control promises 61
5.1.1 bundlesequence 61
5.1.2 goal_patterns 63
5.1.3 ignore_missing_bundles 63
5.1.4 ignore_missing_inputs 64
5.1.5 inputs 64
5.1.6 version 65
5.1.7 lastseenexpireafter 65
5.1.8 output_prefix 66
5.1.9 domain 66
5.1.10 require_comments 67
5.1.11 host_licenses_paid 67
5.1.12 site_classes 68
5.1.13 syslog_host 68
5.1.14 syslog_port 69
5.1.15 fips_mode 69
5.2 agent control promises 70
5.2.1 abortclasses 70

5.2.2 abortbundleclasses 71
5.2.3 addclasses 72
5.2.4 agentaccess 73
5.2.5 agentfacility 73
5.2.6 allclassesreport 74
5.2.7 alwaysvalidate 74
5.2.8 auditing 75
5.2.9 binarypaddingchar 76
5.2.10 bindtointerface 76
5.2.11 hashupdates 77
5.2.12 childlibpath 77
5.2.13 checksum_alert_time 78
5.2.14 defaultcopytype 78
5.2.15 dryrun 79
5.2.16 editbinaryfilesize 79
5.2.17 editfilesize 80
5.2.18 environment 80
5.2.19 exclamation 81
5.2.20 expireafter 82
5.2.21 files_single_copy 82
5.2.22 files_auto_define 82
5.2.23 hostnamekeys 83
5.2.24 ifelapsed 84
5.2.25 inform 84
5.2.26 intermittency 85
5.2.27 max_children 85
5.2.28 maxconnections 86
5.2.29 mountfilesystems 87
5.2.30 nonalphanumfiles 87
5.2.31 repchar 88
5.2.32 refresh_processes 88
5.2.33 default_repository 89
5.2.34 secureinput 89
5.2.35 sensiblecount 90
5.2.36 sensiblesize 90
5.2.37 skipidentify 91
5.2.38 suspiciousnames 91
5.2.39 syslog 92
5.2.40 track_value 92
5.2.41 timezone 93
5.2.42 default_timeout 93
5.2.43 verbose 94
5.3 server control promises 94
5.3.1 allowallconnects 95
5.3.2 allowconnects 95
5.3.3 allowusers 96
5.3.4 auditing 96
5.3.5 bindtointerface 97

5.3.6 cfruncommand ... 97
5.3.7 call_collect_interval ... 98
5.3.8 collect_window ... 100
5.3.9 denybadclocks ... 100
5.3.10 denyconnects ... 101
5.3.11 dynamicaddresses ... 101
5.3.12 hostnamekeys ... 102
5.3.13 keycacheTTL ... 102
5.3.14 logallconnections ... 103
5.3.15 logencryptedtransfers ... 103
5.3.16 maxconnections ... 104
5.3.17 port ... 104
5.3.18 serverfacility ... 105
5.3.19 skipverify ... 106
5.3.20 trustkeysfrom ... 106
5.3.21 listen ... 107
5.4 monitor control promises ... 107
5.4.1 forgetrate ... 108
5.4.2 monitorfacility ... 108
5.4.3 histograms ... 109
5.4.4 tcpdump ... 110
5.4.5 tcpdumpcommand ... 110
5.5 runagent control promises ... 111
5.5.1 hosts ... 111
5.5.2 port ... 112
5.5.3 force_ipv4 ... 112
5.5.4 trustkey ... 113
5.5.5 encrypt ... 114
5.5.6 background_children ... 114
5.5.7 max_children ... 115
5.5.8 output_to_file ... 115
5.5.9 output_directory ... 116
5.5.10 timeout ... 116
5.6 executor control promises ... 117
5.6.1 splaytime ... 117
5.6.2 mailfrom ... 118
5.6.3 mailto ... 118
5.6.4 smtpserver ... 119
5.6.5 mailmaxlines ... 119
5.6.6 schedule ... 120
5.6.7 executorfacility ... 120
5.6.8 exec_command ... 121
5.7 knowledge control promises ... 121
5.7.1 build_directory ... 122
5.7.2 document_root ... 122
5.7.3 generate_manual ... 123
5.7.4 graph_directory ... 123
5.7.5 graph_output ... 124

5.7.6 html_banner ... 124
5.7.7 html_footer ... 125
5.7.8 id_prefix ... 125
5.7.9 manual_source_directory ... 126
5.7.10 query_engine ... 126
5.7.11 query_output ... 127
5.7.12 sql_type ... 127
5.7.13 sql_database ... 128
5.7.14 sql_owner ... 128
5.7.15 sql_passwd ... 129
5.7.16 sql_server ... 129
5.7.17 sql_connection_db ... 129
5.7.18 style_sheet ... 130
5.7.19 view_projections ... 130
5.8 reporter control promises ... 131
5.8.1 aggregation_point ... 131
5.8.2 auto_scaling ... 132
5.8.3 build_directory ... 132
5.8.4 csv2xml ... 133
5.8.5 error_bars ... 134
5.8.6 html_banner ... 134
5.8.7 html_embed ... 135
5.8.8 html_footer ... 135
5.8.9 query_engine ... 136
5.8.10 reports ... 136
5.8.11 report_output ... 137
5.8.12 style_sheet ... 138
5.8.13 time_stamps ... 138
5.9 hub control promises ... 139
5.9.1 export_zenoss ... 139
5.9.2 exclude_hosts ... 140
5.9.3 hub_schedule ... 140
5.9.4 port ... 141
5.10 file control promises ... 141
5.10.1 namespace ... 142

6 Bundles of common ... 143
6.1 classes promises in '*' ... 143
6.1.1 and ... 144
6.1.2 dist ... 144
6.1.3 expression ... 145
6.1.4 or ... 145
6.1.5 persistence ... 145
6.1.6 not ... 147
6.1.7 select_class ... 147
6.1.8 xor ... 148
6.2 defaults promises in '*' ... 149
6.2.1 if_match_regex ... 151

6.2.2 string 151
6.2.3 slist 152
6.3 meta promises in '*' 152
6.3.1 string 153
6.3.2 slist 153
6.4 reports promises in '*' 154
6.4.1 friend_pattern 155
6.4.2 intermittency 155
6.4.3 lastseen 156
6.4.4 printfile (body template) 156
6.4.5 report_to_file 157
6.4.6 bundle_return_value_index 158
6.4.7 showstate 159
6.5 vars promises in '*' 162
6.5.1 string 162
6.5.2 int 163
6.5.3 real 164
6.5.4 slist 164
6.5.5 ilist 165
6.5.6 rlist 165
6.5.7 policy 166
6.6 * promises 167
6.6.1 action (body template) 167
6.6.2 classes (body template) 179
6.6.3 comment 187
6.6.4 depends_on 187
6.6.5 handle 188
6.6.6 ifvarclass 189
6.6.7 meta 190

7 Bundles of agent 193
7.1 commands promises in 'agent' 193
7.1.1 args 195
7.1.2 contain (body template) 195
7.1.3 module 200
7.2 databases promises in 'agent' 203
7.2.1 database_server (body template) 206
7.2.2 database_type 208
7.2.3 database_operation 208
7.2.4 database_columns 208
7.2.5 database_rows 209
7.2.6 registry_exclude 210
7.3 guest_environments promises in 'agent' 211
7.3.1 environment_host 211
7.3.2 environment_interface (body template) 212
7.3.3 environment_resources (body template) 214
7.3.4 environment_state 216
7.3.5 environment_type 217

7.4 files promises in 'agent' 218
7.4.1 acl (body template) 226
7.4.2 changes (body template) 230
7.4.3 copy_from (body template) 233
7.4.4 create 246
7.4.5 delete (body template) 247
7.4.6 depth_search (body template) 249
7.4.7 edit_defaults (body template) 253
7.4.8 edit_line 257
7.4.9 edit_template 257
7.4.10 edit_xml 258
7.4.11 file_select (body template) 258
7.4.12 link_from (body template) 267
7.4.13 move_obstructions 271
7.4.14 pathtype 271
7.4.15 perms (body template) 273
7.4.16 rename (body template) 275
7.4.17 repository 277
7.4.18 touch 278
7.4.19 transformer 279
7.5 Miscelleneous in edit_line promises 280
7.5.1 select_region (body template) 280
7.6 delete_lines promises in 'edit_line' 283
7.6.1 delete_select (body template) 284
7.6.2 not_matching 287
7.7 insert_lines promises in 'edit_line' 288
7.7.1 expand_scalars 290
7.7.2 insert_type 292
7.7.3 insert_select (body template) 293
7.7.4 location (body template) 296
7.7.5 whitespace_policy 298
7.8 field_edits promises in 'edit_line' 299
7.8.1 edit_field (body template) 301
7.9 replace_patterns promises in 'edit_line' 305
7.9.1 replace_with (body template) 306
7.10 Miscelleneous in edit_xml promises 307
7.10.1 build_xpath 307
7.10.2 select_xpath 308
7.11 build_xpath promises in 'edit_xml' 308
7.12 delete_tree promises in 'edit_xml' 309
7.13 insert_tree promises in 'edit_xml' 309
7.14 delete_attribute promises in 'edit_xml' 310
7.15 set_attribute promises in 'edit_xml' 310
7.15.1 attribute_value 311
7.16 delete_text promises in 'edit_xml' 311
7.17 set_text promises in 'edit_xml' 312
7.18 insert_text promises in 'edit_xml' 312
7.19 interfaces promises in 'agent' 313

7.19.1 tcp_ip (body template) 313
7.20 methods promises in 'agent' 315
7.20.1 inherit 316
7.20.2 usebundle 317
7.20.3 useresult 317
7.21 outputs promises in 'agent' 318
7.21.1 output_level 320
7.21.2 promiser_type 320
7.22 packages promises in 'agent' 321
7.22.1 package_architectures 325
7.22.2 package_method (body template) 325
7.22.3 package_policy 341
7.22.4 package_select 342
7.22.5 package_version 343
7.23 processes promises in 'agent' 343
7.23.1 process_count (body template) 347
7.23.2 process_select (body template) 349
7.23.3 process_stop 354
7.23.4 restart_class 355
7.23.5 signals 355
7.24 services promises in 'agent' 357
7.24.1 service_policy 359
7.24.2 service_dependencies 359
7.24.3 service_method (body template) 360
7.25 storage promises in 'agent' 363
7.25.1 mount (body template) 364
7.25.2 volume (body template) 367

8 Bundles of server 371
8.1 access promises in 'server' 371
8.1.1 admit 373
8.1.2 deny 374
8.1.3 maproot 374
8.1.4 ifencrypted 375
8.1.5 resource_type 376
8.2 roles promises in 'server' 377
8.2.1 authorize 378

9 Bundles of knowledge 381
9.1 inferences promises in 'knowledge' 381
9.1.1 precedents 382
9.1.2 qualifiers 382
9.2 things promises in 'knowledge' 383
9.2.1 synonyms 384
9.2.2 affects 384
9.2.3 belongs_to 384
9.2.4 causes 385
9.2.5 certainty 385

9.2.6 determines 386
9.2.7 generalizations 386
9.2.8 implements 387
9.2.9 involves 387
9.2.10 is_caused_by 388
9.2.11 is_connected_to 388
9.2.12 is_determined_by 389
9.2.13 is_followed_by 389
9.2.14 is_implemented_by 390
9.2.15 is_located_in 390
9.2.16 is_measured_by 391
9.2.17 is_part_of 391
9.2.18 is_preceded_by 391
9.2.19 measures 392
9.2.20 needs 392
9.2.21 provides 393
9.2.22 uses 393
9.3 topics promises in 'knowledge' 393
9.3.1 association (body template) 394
9.3.2 synonyms 396
9.3.3 generalizations 396
9.4 occurrences promises in 'knowledge' 397
9.4.1 about_topics 397
9.4.2 represents 398
9.4.3 representation 398

10 Bundles of monitor 401
10.1 measurements promises in 'monitor' 401
10.1.1 stream_type 404
10.1.2 data_type 405
10.1.3 history_type 406
10.1.4 units 406
10.1.5 match_value (body template) 407

11 Special functions 411
11.1 Introduction to functions 411
11.1.1 Functions listed by return value 411
Functions which return class 411
Functions which return (i,r,s)list 411
Functions which return int 412
Functions which return (i,r)range 412
Functions which return real 412
Functions which return string 412
11.1.2 Functions which fill arrays 412
11.1.3 Functions which read "large" data 413
Functions which read arrays 413
Functions which read disk data 413
Functions which read from a remote-CFEngine 413

Functions which read classes 413
Functions which read command output 413
Functions which read the environment 413
Functions which read files 413
Functions which read LDAP data 414
Functions which read from the network 414
Functions which read the Windows registry 414
Functions which read (i,r,s)lists 414
Functions which read strings 414
11.1.4 Functions which look at file metadata 414
11.1.5 Functions which look at variables 415
11.1.6 Functions involving date or time 415
11.1.7 Functions which work with or on regular expressions 415
11.2 Function accessedbefore 415
11.3 Function accumulated 416
11.4 Function ago 418
11.5 Function and 419
11.6 Function canonify 420
11.7 Function concat 420
11.8 Function changedbefore 421
11.9 Function classify 422
11.10 Function classmatch 422
11.11 Function countclassesmatching 423
11.12 Function countlinesmatching 424
11.13 Function dirname 424
11.14 Function diskfree 425
11.15 Function escape 426
11.16 Function execresult 426
11.17 Function fileexists 427
11.18 Function filesexist 428
11.19 Function filesize 429
11.20 Function getenv 430
11.21 Function getfields 431
11.22 Function getgid 432
11.23 Function getindices 433
11.24 Function getuid 434
11.25 Function getusers 435
11.26 Function getvalues 435
11.27 Function grep 436
11.28 Function groupexists 437
11.29 Function hash 438
11.30 Function hashmatch 439
11.31 Function host2ip 440
11.32 Function ip2host 440
11.33 Function hostinnetgroup 441
11.34 Function hostrange 441
11.35 Function hostsseen 442
11.36 Function hostswithclass 443

11.37 Function hubknowledge . . . 444
11.38 Function iprange . . . 445
11.39 Function irange . . . 446
11.40 Function isdir . . . 446
11.41 Function isexecutable . . . 447
11.42 Function isgreaterthan . . . 447
11.43 Function islessthan . . . 448
11.44 Function islink . . . 449
11.45 Function isnewerthan . . . 450
11.46 Function isplain . . . 451
11.47 Function isvariable . . . 452
11.48 Function join . . . 453
11.49 Function lastnode . . . 454
11.50 Function laterthan . . . 454
11.51 Function ldaparray . . . 455
11.52 Function ldaplist . . . 456
11.53 Function ldapvalue . . . 457
11.54 Function lsdir . . . 458
11.55 Function maplist . . . 459
11.56 Function not . . . 460
11.57 Function now . . . 460
11.58 Function on . . . 460
11.59 Function or . . . 461
11.60 Function parseintarray . . . 462
11.61 Function parserealarray . . . 463
11.62 Function parsestringarray . . . 464
11.63 Function parsestringarrayidx . . . 465
11.64 Function peers . . . 466
11.65 Function peerleader . . . 468
11.66 Function peerleaders . . . 469
11.67 Function product . . . 471
11.68 Function randomint . . . 472
11.69 Function readfile . . . 472
11.70 Function readintarray . . . 473
11.71 Function readintlist . . . 474
11.72 Function readrealarray . . . 475
11.73 Function readreallist . . . 476
11.74 Function readstringarray . . . 477
11.75 Function readstringarrayidx . . . 479
11.76 Function readstringlist . . . 480
11.77 Function readtcp . . . 482
11.78 Function regarray . . . 483
11.79 Function regcmp . . . 484
11.80 Function regextract . . . 486
11.81 Function registryvalue . . . 487
11.82 Function regline . . . 488
11.83 Function reglist . . . 489
11.84 Function regldap . . . 490

11.85 Function remotescalar 491
11.86 Function remoteclassesmatching 493
11.87 Function returnszero 493
11.88 Function rrange 494
11.89 Function selectservers 495
11.90 Function splayclass 497
11.91 Function splitstring 498
11.92 Function strcmp 499
11.93 Function sum 500
11.94 Function translatepath 501
11.95 Function usemodule 502
11.96 Function userexists 503

12 Special Variables 505
12.1 Variable context `const` 505
12.1.1 Variable const.dollar 505
12.1.2 Variable const.endl 505
12.1.3 Variable const.n 505
12.1.4 Variable const.r 506
12.1.5 Variable const.t 506
12.2 Variable context `edit` 506
12.2.1 Variable edit.filename 507
12.3 Variable context `match` 507
12.3.1 Variable match.0 507
12.4 Variable context `mon` 508
12.4.1 Variable mon.listening_udp4_ports 508
12.4.2 Variable mon.listening_tcp4_ports 508
12.4.3 Variable mon.listening_udp6_ports 508
12.4.4 Variable mon.listening_tcp6_ports 508
12.4.5 Variable mon.value_users 508
12.4.6 Variable mon.av_users 508
12.4.7 Variable mon.dev_users 508
12.4.8 Variable mon.value_rootprocs 509
12.4.9 Variable mon.av_rootprocs 509
12.4.10 Variable mon.dev_rootprocs 509
12.4.11 Variable mon.value_otherprocs 509
12.4.12 Variable mon.av_otherprocs 509
12.4.13 Variable mon.dev_otherprocs 509
12.4.14 Variable mon.value_diskfree 509
12.4.15 Variable mon.av_diskfree 509
12.4.16 Variable mon.dev_diskfree 509
12.4.17 Variable mon.value_loadavg 509
12.4.18 Variable mon.av_loadavg 509
12.4.19 Variable mon.dev_loadavg 510
12.4.20 Variable mon.value_netbiosns_in 510
12.4.21 Variable mon.av_netbiosns_in 510
12.4.22 Variable mon.dev_netbiosns_in 510
12.4.23 Variable mon.value_netbiosns_out 510

12.4.24 Variable mon.av_netbiosns_out . . . 510
12.4.25 Variable mon.dev_netbiosns_out . . . 510
12.4.26 Variable mon.value_netbiosdgm_in . . . 510
12.4.27 Variable mon.av_netbiosdgm_in . . . 510
12.4.28 Variable mon.dev_netbiosdgm_in . . . 510
12.4.29 Variable mon.value_netbiosdgm_out . . . 510
12.4.30 Variable mon.av_netbiosdgm_out . . . 511
12.4.31 Variable mon.dev_netbiosdgm_out . . . 511
12.4.32 Variable mon.value_netbiosssn_in . . . 511
12.4.33 Variable mon.av_netbiosssn_in . . . 511
12.4.34 Variable mon.dev_netbiosssn_in . . . 511
12.4.35 Variable mon.value_netbiosssn_out . . . 511
12.4.36 Variable mon.av_netbiosssn_out . . . 511
12.4.37 Variable mon.dev_netbiosssn_out . . . 511
12.4.38 Variable mon.value_imap_in . . . 511
12.4.39 Variable mon.av_imap_in . . . 511
12.4.40 Variable mon.dev_imap_in . . . 511
12.4.41 Variable mon.value_imap_out . . . 512
12.4.42 Variable mon.av_imap_out . . . 512
12.4.43 Variable mon.dev_imap_out . . . 512
12.4.44 Variable mon.value_cfengine_in . . . 512
12.4.45 Variable mon.av_cfengine_in . . . 512
12.4.46 Variable mon.dev_cfengine_in . . . 512
12.4.47 Variable mon.value_cfengine_out . . . 512
12.4.48 Variable mon.av_cfengine_out . . . 512
12.4.49 Variable mon.dev_cfengine_out . . . 512
12.4.50 Variable mon.value_nfsd_in . . . 512
12.4.51 Variable mon.av_nfsd_in . . . 512
12.4.52 Variable mon.dev_nfsd_in . . . 513
12.4.53 Variable mon.value_nfsd_out . . . 513
12.4.54 Variable mon.av_nfsd_out . . . 513
12.4.55 Variable mon.dev_nfsd_out . . . 513
12.4.56 Variable mon.value_smtp_in . . . 513
12.4.57 Variable mon.av_smtp_in . . . 513
12.4.58 Variable mon.dev_smtp_in . . . 513
12.4.59 Variable mon.value_smtp_out . . . 513
12.4.60 Variable mon.av_smtp_out . . . 513
12.4.61 Variable mon.dev_smtp_out . . . 513
12.4.62 Variable mon.value_www_in . . . 513
12.4.63 Variable mon.av_www_in . . . 514
12.4.64 Variable mon.dev_www_in . . . 514
12.4.65 Variable mon.value_www_out . . . 514
12.4.66 Variable mon.av_www_out . . . 514
12.4.67 Variable mon.dev_www_out . . . 514
12.4.68 Variable mon.value_ftp_in . . . 514
12.4.69 Variable mon.av_ftp_in . . . 514
12.4.70 Variable mon.dev_ftp_in . . . 514
12.4.71 Variable mon.value_ftp_out . . . 514

12.4.72 Variable mon.av_ftp_out 514
12.4.73 Variable mon.dev_ftp_out 514
12.4.74 Variable mon.value_ssh_in 515
12.4.75 Variable mon.av_ssh_in 515
12.4.76 Variable mon.dev_ssh_in 515
12.4.77 Variable mon.value_ssh_out 515
12.4.78 Variable mon.av_ssh_out 515
12.4.79 Variable mon.dev_ssh_out 515
12.4.80 Variable mon.value_wwws_in 515
12.4.81 Variable mon.av_wwws_in 515
12.4.82 Variable mon.dev_wwws_in 515
12.4.83 Variable mon.value_wwws_out 515
12.4.84 Variable mon.av_wwws_out 515
12.4.85 Variable mon.dev_wwws_out 516
12.4.86 Variable mon.value_icmp_in 516
12.4.87 Variable mon.av_icmp_in 516
12.4.88 Variable mon.dev_icmp_in 516
12.4.89 Variable mon.value_icmp_out 516
12.4.90 Variable mon.av_icmp_out 516
12.4.91 Variable mon.dev_icmp_out 516
12.4.92 Variable mon.value_udp_in 516
12.4.93 Variable mon.av_udp_in 516
12.4.94 Variable mon.dev_udp_in 516
12.4.95 Variable mon.value_udp_out 516
12.4.96 Variable mon.av_udp_out 516
12.4.97 Variable mon.dev_udp_out 517
12.4.98 Variable mon.value_dns_in 517
12.4.99 Variable mon.av_dns_in 517
12.4.100 Variable mon.dev_dns_in 517
12.4.101 Variable mon.value_dns_out 517
12.4.102 Variable mon.av_dns_out 517
12.4.103 Variable mon.dev_dns_out 517
12.4.104 Variable mon.value_tcpsyn_in 517
12.4.105 Variable mon.av_tcpsyn_in 517
12.4.106 Variable mon.dev_tcpsyn_in 517
12.4.107 Variable mon.value_tcpsyn_out 517
12.4.108 Variable mon.av_tcpsyn_out 517
12.4.109 Variable mon.dev_tcpsyn_out 518
12.4.110 Variable mon.value_tcpack_in 518
12.4.111 Variable mon.av_tcpack_in 518
12.4.112 Variable mon.dev_tcpack_in 518
12.4.113 Variable mon.value_tcpack_out 518
12.4.114 Variable mon.av_tcpack_out 518
12.4.115 Variable mon.dev_tcpack_out 518
12.4.116 Variable mon.value_tcpfin_in 518
12.4.117 Variable mon.av_tcpfin_in 518
12.4.118 Variable mon.dev_tcpfin_in 518
12.4.119 Variable mon.value_tcpfin_out 518

12.4.120 Variable mon.av_tcpfin_out 518
12.4.121 Variable mon.dev_tcpfin_out 518
12.4.122 Variable mon.value_tcpmisc_in 518
12.4.123 Variable mon.av_tcpmisc_in 519
12.4.124 Variable mon.dev_tcpmisc_in 519
12.4.125 Variable mon.value_tcpmisc_out 519
12.4.126 Variable mon.av_tcpmisc_out 519
12.4.127 Variable mon.dev_tcpmisc_out 519
12.4.128 Variable mon.value_webaccess 519
12.4.129 Variable mon.av_webaccess 519
12.4.130 Variable mon.dev_webaccess 519
12.4.131 Variable mon.value_weberrors 519
12.4.132 Variable mon.av_weberrors 519
12.4.133 Variable mon.dev_weberrors 519
12.4.134 Variable mon.value_syslog 519
12.4.135 Variable mon.av_syslog 519
12.4.136 Variable mon.dev_syslog 519
12.4.137 Variable mon.value_messages 520
12.4.138 Variable mon.av_messages 520
12.4.139 Variable mon.dev_messages 520
12.4.140 Variable mon.value_temp0 520
12.4.141 Variable mon.av_temp0 520
12.4.142 Variable mon.dev_temp0 520
12.4.143 Variable mon.value_temp1 520
12.4.144 Variable mon.av_temp1 520
12.4.145 Variable mon.dev_temp1 520
12.4.146 Variable mon.value_temp2 520
12.4.147 Variable mon.av_temp2 520
12.4.148 Variable mon.dev_temp2 521
12.4.149 Variable mon.value_temp3 521
12.4.150 Variable mon.av_temp3 521
12.4.151 Variable mon.dev_temp3 521
12.4.152 Variable mon.value_cpu 521
12.4.153 Variable mon.av_cpu 521
12.4.154 Variable mon.dev_cpu 521
12.4.155 Variable mon.value_cpu0 521
12.4.156 Variable mon.av_cpu0 521
12.4.157 Variable mon.dev_cpu0 521
12.4.158 Variable mon.value_cpu1 521
12.4.159 Variable mon.av_cpu1 522
12.4.160 Variable mon.dev_cpu1 522
12.4.161 Variable mon.value_cpu2 522
12.4.162 Variable mon.av_cpu2 522
12.4.163 Variable mon.dev_cpu2 522
12.4.164 Variable mon.value_cpu3 522
12.4.165 Variable mon.av_cpu3 522
12.4.166 Variable mon.dev_cpu3 522
12.4.167 Variable mon.value_microsoft_ds_in 522

12.4.168 Variable mon.av_microsoft_ds_in ... 522
12.4.169 Variable mon.dev_microsoft_ds_in ... 522
12.4.170 Variable mon.value_microsoft_ds_out ... 523
12.4.171 Variable mon.av_microsoft_ds_out ... 523
12.4.172 Variable mon.dev_microsoft_ds_out ... 523
12.4.173 Variable mon.value_www_alt_in ... 523
12.4.174 Variable mon.av_www_alt_in ... 523
12.4.175 Variable mon.dev_www_alt_in ... 523
12.4.176 Variable mon.value_www_alt_out ... 523
12.4.177 Variable mon.av_www_alt_out ... 523
12.4.178 Variable mon.dev_www_alt_out ... 523
12.4.179 Variable mon.value_imaps_in ... 523
12.4.180 Variable mon.av_imaps_in ... 523
12.4.181 Variable mon.dev_imaps_in ... 524
12.4.182 Variable mon.value_imaps_out ... 524
12.4.183 Variable mon.av_imaps_out ... 524
12.4.184 Variable mon.dev_imaps_out ... 524
12.4.185 Variable mon.value_ldap_in ... 524
12.4.186 Variable mon.av_ldap_in ... 524
12.4.187 Variable mon.dev_ldap_in ... 524
12.4.188 Variable mon.value_ldap_out ... 524
12.4.189 Variable mon.av_ldap_out ... 524
12.4.190 Variable mon.dev_ldap_out ... 524
12.4.191 Variable mon.value_ldaps_in ... 524
12.4.192 Variable mon.av_ldaps_in ... 525
12.4.193 Variable mon.dev_ldaps_in ... 525
12.4.194 Variable mon.value_ldaps_out ... 525
12.4.195 Variable mon.av_ldaps_out ... 525
12.4.196 Variable mon.dev_ldaps_out ... 525
12.4.197 Variable mon.value_mongo_in ... 525
12.4.198 Variable mon.av_mongo_in ... 525
12.4.199 Variable mon.dev_mongo_in ... 525
12.4.200 Variable mon.value_mongo_out ... 525
12.4.201 Variable mon.av_mongo_out ... 525
12.4.202 Variable mon.dev_mongo_out ... 525
12.4.203 Variable mon.value_mysql_in ... 526
12.4.204 Variable mon.av_mysql_in ... 526
12.4.205 Variable mon.dev_mysql_in ... 526
12.4.206 Variable mon.value_mysql_out ... 526
12.4.207 Variable mon.av_mysql_out ... 526
12.4.208 Variable mon.dev_mysql_out ... 526
12.4.209 Variable mon.value_postgres_in ... 526
12.4.210 Variable mon.av_postgres_in ... 526
12.4.211 Variable mon.dev_postgres_in ... 526
12.4.212 Variable mon.value_postgres_out ... 526
12.4.213 Variable mon.av_postgres_out ... 526
12.4.214 Variable mon.dev_postgres_out ... 527
12.4.215 Variable mon.value_ipp_in ... 527

12.4.216 Variable mon.av_ipp_in 527
12.4.217 Variable mon.dev_ipp_in 527
12.4.218 Variable mon.value_ipp_out 527
12.4.219 Variable mon.av_ipp_out 527
12.4.220 Variable mon.dev_ipp_out 527
12.5 Variable context sys 527
12.5.1 Variable sys.arch 528
12.5.2 Variable sys.cdate 528
12.5.3 Variable sys.cf_version 528
12.5.4 Variable sys.class 528
12.5.5 Variable sys.cpus 528
12.5.6 Variable sys.crontab 529
12.5.7 Variable sys.date 529
12.5.8 Variable sys.doc_root 529
12.5.9 Variable sys.domain 529
12.5.10 Variable sys.enterprise_version 529
12.5.11 Variable sys.expires 530
12.5.12 Variable sys.exports 530
12.5.13 Variable sys.flavor 530
12.5.14 Variable sys.flavour 530
12.5.15 Variable sys.fqhost 530
12.5.16 Variable sys.fstab 531
12.5.17 Variable sys.hardware_addresses 531
12.5.18 Variable sys.hardware_mac[interface_name] 531
12.5.19 Variable sys.host 531
12.5.20 Variable sys.interface 531
12.5.21 Variable sys.interfaces 532
12.5.22 Variable sys.ip_addresses 532
12.5.23 Variable sys.ipv4 533
12.5.24 Variable sys.ipv4[interface_name] 533
12.5.25 Variable sys.ipv4_1[interface_name] 534
12.5.26 Variable sys.ipv4_2[interface_name] 534
12.5.27 Variable sys.ipv4_3[interface_name] 534
12.5.28 Variable sys.license_owner 534
12.5.29 Variable sys.licenses_granted 534
12.5.30 Variable sys.licenses_installtime 535
12.5.31 Variable sys.long_arch 535
12.5.32 Variable sys.maildir 535
12.5.33 Variable sys.nova_version 535
12.5.34 Variable sys.os 535
12.5.35 Variable sys.ostype 536
12.5.36 Variable sys.policy_hub 536
12.5.37 Variable sys.release 536
12.5.38 Variable sys.resolv 536
12.5.39 Variable sys.uqhost 536
12.5.40 Variable sys.version 537
12.5.41 Variable sys.windir 537
12.5.42 Variable sys.winprogdir 537

12.5.43 Variable sys.winprogdir86 537
12.5.44 Variable sys.winsysdir 537
12.5.45 Variable sys.workdir 538
12.6 Variable context `this` 538
12.6.1 Variable this.handle 538
12.6.2 Variable this.promise_filename 539
12.6.3 Variable this.promise_linenumber 539
12.6.4 Variable this.promiser 539
12.6.5 Variable service_policy 539
12.6.6 Variable this.this 539

13 Logs and records 541
13.1 Embedded Databases 541
13.2 Text logs 542
13.3 Reports in outputs 543
13.4 Additional reports in commcerical CFEngine versions 543
13.5 State information 543

1 CFEngine 3.5.0a1.b704ecb – Getting started

CFEngine is a suite of programs for integrated autonomic management of either individual or networked computers. It has existed as as software suite since 1993 and this version published under the GNU Public License (GPL v3) and a Commercial Open Source License (COSL). CFEngine is Copyright by **CFEngine AS**, a company founded by CFEngine author Mark Burgess.

This document describes major version 3 of CFEngine, which is a significant departure from earlier versions, and represents the newest and most carefully researched technology available for configuration management. It is both simpler and more powerful. CFEngine 3 will exist as both free open source and commercial enterprise versions:

- **Community Edition** - a free and gratis core of the software (available now).
- **Enterprise** - a commercial enhanced version for basic enterprise needs (available now; previously known as Nova).

This document is valid for **all versions** of CFEngine. Whenever a feature is only available in a specific version, that fact will be noted in the documentation for that feature (if there is no note, then that feature is available in all versions).

CFEngine 3 has been changed to be both a more powerful tool and a much simpler tool. CFEngine 3's language interface is not backwards compatible with the CFEngine 2 configuration language, but it interoperates with CFEngine 2 so that it is "run-time compatible". This means that you can change over to version 3 slowly, with low risk and at your own speed.

With CFEngine 3 you can install, configure and maintain computers using powerful hands-free tools. You can also integrate knowledge management and diagnosis into the processes.

CFEngine differs from most management systems in being

- Open software (GPL or COSL).
- Lightweight and generic.
- Non-reliant on a working network to function correctly.
- Capable of making each and every host autonomous

1.1 Software components

CFEngine 3 consists of a number of components. The names of the programs are intentionally different from those in CFEngine 2 to help disambiguate them (and some CFEngine 2 components have been merged and/or eliminated). The starred components are new to CFEngine 3:

1.1.1 cf-agent

Active agent – responsible for maintaining promises about the state of your system (in CFEngine 2 the agent was called `cfagent`). You can run `cf-agent` manually, but if you want to have it run on a regular basis, you should use Section 1.1.2 [cf-execd], page 2 (instead of using `cron`).

`cf-agent` keeps the promises made in Chapter 6 [common], page 143 and Chapter 7 [agent], page 193 bundles, and is affected by Section 5.1 [common], page 61 and Section 5.2 [agent], page 70 control bodies.

1.1.2 cf-execd

Scheduler – responsible for running cf-agent on a regular (and user-configurable) basis (in CFEngine 2 the scheduler was called `cfexecd`).

EXECUTOR `cf-execd` keeps the promises made in Chapter 6 [common], page 143 bundles, and is affected by Section 5.1 [common], page 61 and Section 5.6 [executor], page 117 control bodies.

1.1.3 cf-know*

Knowledge modelling agent – responsible for building and analysing a semantic knowledge network.

`cf-know` keeps the promises made in Chapter 6 [common], page 143 and Chapter 9 [knowledge], page 381 bundles, and is affected by Section 5.1 [common], page 61 and Section 5.7 [knowledge], page 121 control bodies.

1.1.4 cf-monitord

Passive monitoring agent – responsible for collecting information about the status of your system (which can be reported upon or used to enforce promises or influence when promises are enforced). In CFEngine 2 the passive monitoring agent was known as `cfenvd`.

`cf-monitord` keeps the promises made in Chapter 6 [common], page 143 and Chapter 10 [monitor], page 401 bundles, and is affected by Section 5.1 [common], page 61 and Section 5.4 [monitor], page 107 control bodies.

1.1.5 cf-promises

Promise validator – used to verify that the promises used by the other components of CFEngine are syntactically valid. `cf-promises` does not execute any promises, but can syntax-check all of them.

1.1.6 cf-runagent

Remote run agent – used to execute `cf-agent` on a remote machine (in CFEngine 2 the remote run agent was called `cfrun`). `cf-runagent` does not keep any promises, but instead is used to ask another machine to do so.

1.1.7 cf-serverd

Server – used to distribute policy and/or data files to clients requesting them and used to respond to requests from `cf-runagent` (in CFEngine 2 the remote run agent was called `cfservd`).

`cf-serverd` keeps the promises made in Chapter 6 [common], page 143 and Chapter 8 [server], page 371 bundles, and is affected by Section 5.1 [common], page 61 and Section 5.3 [server], page 94 control bodies.

1.1.8 cf-report

Self-knowledge extractor – takes data stored in CFEngine's embedded databases and converts them to human readable form

`cf-report` keeps the promises made in Chapter 6 [common], page 143 bundles, and is affected by Section 5.1 [common], page 61 and Section 5.8 [reporter], page 131 control bodies.

1.1.9 cf-key

Key generation tool – run once on every host to create public/private key pairs for secure communication (in CFEngine 2 the key generation tool was called `cfkey`). `cf-key` does not keep any promises.

1.1.10 cf-hub

A data aggregator used as part of the commercial product. This stub is not used in the community edition of CFEngine.

1.2 Core concepts

Unlike previous versions of CFEngine, which had no consistent model for its features, you can recognize *everything* in CFEngine 3 from just a few concepts.

Promise A declaration about the *state* we desire to maintain (*e.g.,* the permissions or contents of a file, the availability or absence of a service, the (de)installation of a package).

Promise bundles
A collection of promises.

Promise bodies
A part of a promise which details and constrains its nature.

Data types An interpretation of a scalar value: string, integer or real number.

Variables An association of the form "LVALUE *represents* RVALUE", where rval may be a scalar value or a list of scalar values.

Functions Built-in parameterized rvalues.

Classes CFEngine's boolean classifiers that describe context.

If you have used CFEngine before then the most visible part of CFEngine 3 will be its new language interface. Although it has been clear for a long time that the organically grown language used in CFEngine 1 and 2 developed many problems, it was not immediately clear exactly what would be better. It has taken years of research to simplify the successful features of CFEngine to a single overarching model. To understand the new CFEngine, it is best to set aside any preconceptions about what CFEngine is today. CFEngine 3 is a genuine "next generation" effort, which will be a springboard into the future of system management.

1.3 A renewed CFEngine

CFEngine 3 is a significant rewrite of underlying CFEngine technology which preserves the core principles and methodology of CFEngine's tried and tested approach. It comes with a new, improved language, with a consistent syntax and powerful pattern expression features that display the intent behind CFEngine code more clearly. The main goal in changing the language is to simplify and improve the robustness and functionality without sacrificing the basic freedoms and self-repairing concepts.

CFEngine 3's new language is a direct implementation of a model developed at Oslo University College over the past four years, known colloquially as "Promise Theory". Promises were originally introduced by Mark Burgess as a way to talk about CFEngine's model of autonomy and have since become a powerful way of modelling cooperative systems – not just computers, but humans too.

> *"The biggest challenge of implementing CFEngine in our organization*
> *was not technical but political – getting everyone to agree.*
> *Promise theory was a big help in understand this."*

CFEngine 3 is a generic implementation of the language of promises that allows all of the aspects of configuration and change management to be unified under a single umbrella.

Why talk about promises instead of simply talking about changes? After all, the trend in business and IT management today is to talk about Change Management (with capital letters), e.g. in the IT Infrastructure Library (ITIL) terminology. This comes from a long history of process management thinking. But we are not really interested in change – we are interested in avoiding it, i.e. being in a state where we don't need to make any changes. In other words we want to be able to promise that the system is correct, verify this and only make changes if our promises are not kept. If you want to think ITIL, think of this as a service that CFEngine provides.

To put it another way, CFEngine is not really a *change management* system, it is a *maintenance system*. Maintenance is the process of making small changes or corrections to a model. A 'model' is just another word for a template or a specification of how we want the system to work. CFEngine's model is based on the idea of promises, which means that it focuses on what is stable and lasting about a system – not about what is changing.

This is an important philosophical shift. It means we are focused mainly on what is right and not on what is wrong. By saying what "right" is (the ideal state of our system) we are focused on the actual behaviour. If we focus too much on the changes, i.e. the differences between now and the future, we might forget to verify that what we assume is working now in fact works.

Models that talk about change management tend to forget that after every change there is a litany of *incidents* during which it is necessary to repair the system or return it to its intended state. But if we know what we have promised, it is easy to verify whether the promise is kept. This means that it is the *promises* about how the system should be that are most important, not the actual changes that are made in order to keep them.

1.4 Installation

In order to install CFEngine, you should first ensure that the following packages are installed.

OpenSSL Open source Secure Sockets Layer for encryption.
URL: `http://www.openssl.org`

Tokyo Cabinet (version 1.4.42 or later)
Lightweight flat-file database system.
URL: `http://fallabs.com/tokyocabinet/`

PCRE Perl Compatible Regular Expression library.
URL: `http://www.pcre.org/`

On Windows machines, you need to install the basic Cygwin DLL from `http://www.cygwin.com` in order to run CFEngine.

Additional functionality becomes available if other libraries are present, e.g. OpenLDAP, client libraries for MySQL and PostgreSQL, etc. It is possible to run CFEngine without these, but related functionality will be missing.

Unless you have purchased ready-to-run binaries, or are using a package distribution, you will need to compile CFEngine. For this you will also need a build environment tools: `gcc`, `flex`, `bison`.

The preferred method of installation is then

```
tar zxf cfengine-x.x.x.tar.gz
cd cfengine-x.x.x
./configure
make
make install
```

This results in binaries being installed in '`/var/cfengine/bin`'.

1.5 Syntax, identifiers and names

The CFEngine 3 language has a few simple rules:

- CFEngine built-in words, and identifiers of your choosing (the names of variables, bundles, body templates and classes) may only contain the usual alphanumeric and underscore characters (‘a-zA-Z0-9_’).
- All other ‘literal’ data must be quoted.
- Declarations of promise bundles in the form:

```
bundle agent-type identifier
{
...
}
```

- Declarations of promise body-parts in the form:

```
body constraint_type template_identifier
{
...
}
```

  matching and expanding on a reference inside a promise of the form ‘constraint_type => template_identifier’.
- CFEngine uses many ‘constraint expressions’ as part of the body of a promise. These take the form: left-hand-side (CFEngine word) ‘=>’ right-hand-side (user defined data). This can take several forms:

```
cfengine_word => user_defined_template(parameters)
                 user_defined_template
                 builtin_function()
                 "quoted literal scalar"
                 { list }
```

  In each of these cases, the right hand side is a user choice.

1.6 The work directory

In order to achieve the desired simplifications, it was decided to reserve a private work area for the CFEngine tool-set.

> In CFEngine 1.x, the administrator could choose the locations of configuration files, locks, and logging data independently. In CFEngine 2.x, this diversity has been simplified to a single directory which defaults to ‘/var/cfengine’ (similar to ‘/var/cron’), and in CFEngine 3.x this is preserved.

```
/var/cfengine
/var/cfengine/bin
/var/cfengine/inputs
/var/cfengine/outputs
```

A trusted cache of the input files must now be maintained in the ‘inputs’ subdirectory. When CFEngine is invoked by the scheduler, it reads only from this directory. It is up to the user to keep this cache updated, on each host. This simplifies and consolidates the CFEngine resources in a single place.

Unlike CFEngine 2, CFEngine 3 does not recognize the CFINPUTS environment variable.

The 'outputs' directory is now a record of spooled run-reports. These are often mailed to the administrator by cf-execd, or can be copied to another central location and viewed in an alternative browser.

1.7 Decisions

CFEngine decisions are made behind the scenes and the results of certain true/false propositions are cached in Booleans referred to as 'classes'. There are no if-then-else statements in CFEngine; all decisions are made with classes.

Classes fall into hard (discovered) and soft (user-defined) types. A single hard class can be one of several things:

1.7.1 CFEngine hard classes

CFEngine runs on every computer individually and each time it wakes up the underlying generic agent platform discovers and classifies properties of the environment or context in which it runs. This information is cached and may be used to make decisions about configuration[1].

Classes fall into hard (discovered) and soft (defined) types. A single class can be one of several things:

- The name of an operating system architecture e.g. ultrix, sun4, etc.
- The unqualified name of a particular host (e.g., www). If your system returns a fully qualified domain name for your host (e.g., www.iu.hio.no), CFEngine will also define a hard class for the fully qualified name, as well as the partially-qualified component names iu.hio.no, hio.no, and no.
- The name of a user-defined group of hosts.
- A day of the week (in the form Monday, Tuesday, Wednesday, ..).
- An hour of the day, in the current time zone (in the form Hr00, Hr01 ... Hr23).
- An hour of the day GMT (in the form GMT_Hr00, GMT_Hr01 ... GMT_Hr23). This is consistent the world over, in case you need virtual simulteneity of change coordination.
- Minutes in the hour (in the form Min00, Min17 ... Min45).
- A five minute interval in the hour (in the form Min00_05, Min05_10 ... Min55_00).
- A fifteen minute (quarter-hour) interval (in the form Q1, Q2, Q3, Q4).
- An expression of the current quarter hour (in the form Hr12_Q3).
- A day of the month (in the form Day1, Day2, ... Day31).
- A month (in the form January, February, ... December).
- A year (in the form Yr1997, Yr2004).
- A shift in Night,Morning,Afternoon,Evening, which fall into six hour blocks starting at 00:00 hours.
- A 'lifecycle index', which is the year number modulo 3 (in the form Lcycle_0, Lcycle_1, Lcycle_2, used in long term resource memory).
- An arbitrary user-defined string (as specified in the -D command line option, or defined in a classes promise or body, restart_class in a processes promise, etc).

[1] There are no if-then-else statements in CFEngine; all decisions are made with classes.

- The IP address octets of any active interface (in the form `ipv4_192_0_0_1`, `ipv4_192_0_0`, `ipv4_192_0`, `ipv4_192`), provided they are not excluded by a regular expression in the file '`WORKDIR/inputs/ignore_interfaces.rx`'.
- The names of the active interfaces (in the form `net_iface_xl0`, `net_iface_vr0`).
- System status and entropy information reported by `cf-monitord`.
- On Solaris-10 systems, the zone name (in the form `zone_global`, `zone_foo`, `zone_baz`).

To see all of the classes defined on a particular host, run

```
host# cf-promises -v
```

as a privileged user. Note that some of the classes are set only if a trusted link can be established with cf-monitord, i.e. if both are running with privilege, and the '`/var/cfengine/state/env_data`' file is secure. More information about classes can be found in connection with `allclasses`.

1.7.2 Class combination operators and precedence

Classes may be combined with the usual boolean operators, in the usual precedence (AND binds stronger than OR). On addition the dot may be used for AND to improve readability, or imply the interpretation 'subset' or 'subclass'. In order of precedence:

'`()`' The parenthesis group operator.

'`!`' The NOT operator.

'`.`' The AND operator.

'`&`' The AND operator (alternative).

'`|`' The OR operator.

'`||`' The OR operator (alternative).

So the following expression would be only true on Mondays or Wednesdays from 2:00pm to 2:59pm on Windows XP systems:

```
(Monday|Wednesday).Hr14.WinXP::
```

1.7.3 Global and local classes

User defined classes are mostly defined in bundles, but they are used as a signalling mechanism between promises. We'll return to those in a moment.

Classes promises define new classes based on combinations of old ones. This is how to make complex decisions in CFEngine, with readable results. It is like defining aliases for class combinations. Such class 'aliases' may be specified in any kind of bundle. Bundles of type `common` yield classes that are global in scope, whereas in all other bundle types classes are local. Classes are evaluated when the bundle is evaluated (and the bundles are evaluated in the order specified in the `bundlesequence`). Consider the following example.

```
body common control
{
bundlesequence => { "g","tryclasses_1", "tryclasses_2" };
}
```

```
#################################

bundle common g
{
classes:

  "one" expression => "any";

}

#################################

bundle agent tryclasses_1
{
classes:

  "two" expression => "any";
}

#################################

bundle agent tryclasses_2
{
classes:

  "three" expression => "any";

reports:

  one.three.!two::

    "Success";
}
```

Here we see that class '`one`' is global (because it is defined inside the `common` bundle), while classes '`two`' and '`three`' are local (to their respective bundles). The report result 'Success' is therefore true because only '`one`' and '`three`' are in scope (and '`two`' is *not* in scope) inside of the third bundle.

Note that any class promise must have one - and only one - value constraint. That is, you might not leave '`expression`' in the example above or add both '`and`' and '`xor`' constraints to the single promise.

Another type of class definition happens when you define classes based on the outcome of a promise, e.g. to set a class if a promise is repaired, one might write:

```
 "promiser..."

    ...
```

```
classes => if_repaired("signal_class");
```

These classes are global in scope. Finally `restart_class` classes in `processes` are global.

1.8 Filenames and paths

Filenames in Unix-like operating systems use the forward slash '/' character for their directory separator . All references to file locations must be absolute pathnames in CFEngine, i.e. they must begin with a complete specification of which directory they are in. For example:

```
/etc/passwd
/var/cfengine/masterfiles/distfile
```

The only place where it makes sense to refer to a file without a complete directory specification is when searching through directories for different kinds of file, e.g. in pattern matching

```
leaf_name => { "tmp_.*", "output_file", "core" };
```

Here, one can write '`core`' without a path, because one is looking for any file of that name in a number of directories.

The Windows operating systems traditionally use a different filename convention. The following are all valid absolute file names under Windows:

```
c:\winnt
"c:\spaced name"
c:/winnt
/var/cfengine/inputs
//fileserver/share2/dir
```

The 'drive' name "C:" in Windows refers to a partition or device. Unlike Unix, Windows does not integrate these seamlessly into a single file-tree. This is not a valid absolute filename:

```
\var\cfengine\inputs
```

Paths beginning with a backslash are assumed to be win32 paths. They must begin with a drive letter or double-slash server name.

Note in recent versions of Cygwin you can decide to use the `/cygdrive` to specify a path to windows file E.g '`/cygdrive/c/myfile`' means '`c:\myfile`' or you can do it straight away in CFEngine as `c:\myfile`.

1.9 Upgrading from CFEngine 2

CFEngine 3 has a completely new syntax, designed to solve the issues brought up from 15 years of experience with configuration management. Rather than clutter CFEngine 3 with buggy backward-compatability issues, it was decided to make no compromises with CFEngine 3 and instead allow CFEngine 2 and CFEngine 3 to coincide in a cooperative fashion for the foreseeable future. This means that users can upgrade at their own pace, in the classic CFEngine incremental fashion. We expect that CFEngine 2 installations will be around for years to come so this upgrade path seems the most defensible.

The daemons and support services are fully interoperable between CFEngine 2 and CFEngine 3, so it does not matter whether you run `cfservd` (cf2) together with `cf-agent` (cf3) or `cf-serverd` (cf3) together with `cfagent` (cf2). You can change the servers at your own pace.

CFEngine 3's `cf-execd` replaces CFEngine 2's `cfexecd` and it is designed to work optimally with `cf-agent` (cf3). Running this daemon has no consequences for access control, only for scheduling

`cf-agent`. You can (indeed should) replace `cfexecd` with `cf-execd` immediately. You will want to alter your '`crontab`' file to run the new component instead of the old. The sample CFEngine 3 input files asks `cf-agent` to do this automatically, simply replacing the string.

The sample '`inputs`' files supplied with CFEngine 3 contain promises to integrate CFEngine 2 as described. What can you do to upgrade? Here is a simple recipe that assumes you have a standardized CFEngine 2 setup, running `cfexecd` in '`crontabs`' and possibly running `cfservd` and `cfenvd` as daemons.

1. Install the CFEngine 3 software on a host.
2. Go to the '`inputs/`' directory in the source and copy these files to your master update repository, i.e. where you will publish policies for distribution.
3. Remove any self-healing rules to reinstall CFEngine 2, especially rules to add `cfexecd` or `cfagent` to '`crontabs`' etc. CFEngine 3 will handle this from now on and encapsulate old CFEngine 2 scripts.
4. Move to this inputs directory: `cd` *your-path*`/inputs`.
5. Set the location of this master update directory in the '`update.cf`' file to the location of the master directory.
6. Set the email options for the executor in '`promises.cf`'.
7. Run `cf-agent --bootstrap` as the root or privileged user. This will install CFEngine 3 in place of CFEngine 2, integrate your old CFEngine 2 configuration, and warn you about any rules that need to be removed from your old CFEngine configuration.
8. You should now be running CFEngine 3. You can now add new rules to the files in your own time, or convert the old CFEngine 2 rules and gradually comment them out of the CFEngine 2 files.
9. Make sure there are no rules in your old CFEngine 2 configuration to activate CFEngine 2 components, i.e. rules that will fight against CFEngine 3. Then, when you are ready, convert '`cfservd.conf`' into a server bundle e.g. in '`promises.cf`' and remove all rules to run `cfservd` and replace them with rules to run `cf-serverd` at your own pace.

1.10 Testing as a non-privilieged user

One of the practical advantages of CFEngine is that you can test it without the need for root or administrator privileges. This is recommended for all new users of CFEngine 3.

CFEngine operates with the notion of a work-directory. The default work directory for the `root` user is '`/var/cfengine`' (except on Debian Linux and various derivatives which prefer '`/var/lib/cfengine`'). For any other user, the work directory lies in the user's home directory, named '`~/.cfagent`'. CFEngine prefers you to keep certain files here. You should not resist this too strongly or you will make unnecessary trouble for yourself. The decision to have this 'known directory' was made to simplify a lot of configuration.

To test CFEngine as an ordinary user, do the following:

- Compile and make the software.
- Copy the binaries into the work directory:

```
host$ mkdir -p ~/.cfagent/inputs
host$ mkdir -p ~/.cfagent/bin
host$ cd src
host$ cp cf-* ~/.cfagent/bin
host$ cd ../inputs
host$ cp *.cf ~/.cfagent/inputs
```

You can test the software and play with configuration files by editing the basic get-started files directly in the '~/.cfagent/inputs' directory. For example, try the following:

```
host$ ~/.cfagent/bin/cf-promises
host$ ~/.cfagent/bin/cf-promises --verbose
```

This is always the way to start checking a configuration in CFEngine 3. If a configuration does not pass this check/test, you will not be allowed to use it, and 'cf-agent' will look for the file 'failsafe.cf'.

Notice that the CFEngine 3 binaries have slightly different names than the CFEngine 2 binaries. They all start with the 'cf-' prefix.

```
host$ ~/.cfagent/bin/cf-agent
```

1.11 The 'bare necessities' of a CFEngine 3

Here is the simplest 'Hello world' program in CFEngine 3:

```
body common control
{
bundlesequence  => { "test" };
}

bundle agent test
{
reports:

 Yr2009::
    "Hello world";
}
```

If you try to process this using the cf-promises command, you will see something like this:

```
atlas$ ~/LapTop/CFEngine3/trunk/src/cf-promises -r -f ./unit_null_config.cf
Summarizing promises as text to ./unit_null_config.cf.txt
Summarizing promises as html to ./unit_null_config.cf.html
```

The '-r' option produces a report. Examine the files produced:

```
cat ./unit_null_config.cf.txt
firefox ./unit_null_config.cf.html
```

You will see a summary of how CFEngine interprets the files, either in HTML or text. By default, the CFEngine components also dump a debugging file, e.g. 'promise_output_agent.html', 'promise_output_agent.txt' with an expanded view.

1.12 Familiarizing yourself

To familiarize yourself with CFEngine 3, type or paste in the following example text:

```
########################################################
#
# Simple test execution
#
########################################################
```

```
body common control

{
bundlesequence  => { "testbundle"  };
}

########################################################

bundle agent testbundle

{
vars:

  "size" int => "46k";
  "rand" int => randomint("33","$(size)");

commands:

  "/bin/echo"
     args => "Hello world - $(size)/$(rand)",
     contain => standard,
     classes => cdefine("followup","alert");

  followup::

     "/bin/ls"
       contain => standard;

reports:

  alert::

     "What happened?";

}

######################################################################

body contain standard

{
exec_owner => "mark";
useshell => "true";
}

######################################################################
```

```
body classes cdefine(class,alert)

{
promise_repaired => { "$(class)" };
repair_failed => { "$(alert)" };
}
```

This example shows all of the main features of CFEngine: bundles, bodies, control, variables, and promises. To the casual eye it might look complex, but that is because it is explicit about all of the details. Fortunately it is easy to hide many of these details to make the example simpler without sacrificing any functionality.

The first thing to try with this example is to verify it – did we make any mistakes? Are there any inconsistencies? To do this we use the new CFEngine program `cf-promises`. Let's assume that you typed this into a file called '`test.cf`' in the current directory.

```
cf-promises -f ./test.cf
```

If all is well, typing this command shows no output. Try now running the command with verbose output.

```
cf-promises -f ./test.cf -v
```

Now you see a lot of information

```
Reference time set to Sat Aug  2 11:26:06 2008

cf3 CFEngine - 3.0.0
Free Software Foundation 1994-
Donated by Mark Burgess, Oslo University College, Norway
cf3 -------------------------------------------------------------------------
cf3 Host name is: atlas
cf3 Operating System Type is linux
cf3 Operating System Release is 2.6.22.18-0.2-default
cf3 Architecture = x86_64
cf3 Using internal soft-class linux for host linux
cf3 The time is now Sat Aug  2 11:26:06 2008
cf3 -------------------------------------------------------------------------
cf3 Additional hard class defined as: 64_bit
cf3 Additional hard class defined as: linux_2_6_22_18_0_2_default
cf3 Additional hard class defined as: linux_x86_64
cf3 Additional hard class defined as: linux_x86_64_2_6_22_18_0_2_default
cf3 GNU autoconf class from compile time: compiled_on_linux_gnu
cf3 Interface 1: lo
cf3 Trying to locate my IPv6 address
cf3 Looking for environment from cf-monitord...
cf3 Unable to detect environment from cf-monitord
---------------------------------------------------------------------
Loading persistent classes
---------------------------------------------------------------------

---------------------------------------------------------------------
Loaded persistent memory
---------------------------------------------------------------------
cf3   > Parsing file ./test.cf
---------------------------------------------------------------------
Agent's basic classified context
---------------------------------------------------------------------


Defined Classes = ( any Saturday Hr11 Min26 Min25_30 Q2 Hr11_Q2 Day2
```

```
August Yr2008 linux atlas 64_bit linux_2_6_22_18_0_2_default x86_64
linux_x86_64 linux_x86_64_2_6_22_18_0_2_default
linux_x86_64_2_6_22_18_0_2_default__1_SMP_2008_06_09_13_53_20__0200
compiled_on_linux_gnu net_iface_lo )

Negated Classes = ( )

Installable classes = ( )
cf3 Wrote expansion summary to promise_output_common.html
cf3 Inputs are valid
```

The last two lines of this are of interest. Each time a component of CFEngine 3 parses a number of promises, it summarizes the information in an HTML file called generically `promise_output_`*`component-type`*`.html`. In this case the `cf-promises` command represents all possible promises, by the type "common". You can view this output file in a suitable web browser to see exactly what CFEngine has understood by the configuration.

Now that you have verified it, you can execute it. To run this example you need to change the username 'mark' to your own, or obtain root privileges to change to another user. The non-verbose output of the script when run in the CFEngine 3 directory looks something like this:

```
host$ ./cf-agent -f ../tests/units/unit_exec_in_sequence.cf
Q ".../bin/echo Hello": Hello world - 46k/219
 -> Last 1 QUOTEed lines were generated by "/bin/echo Hello world - 46k/219"
Q ".../bin/ls": agent.c
Q ".../bin/ls": agentdiagnostic.c
Q ".../bin/ls": agentdiagnostic.o
Q ".../bin/ls": agent.o
Q ".../bin/ls": args.c
Q ".../bin/ls": args.lo
Q ".../bin/ls": args.o
...
Q ".../bin/ls": verify_reports.o
Q ".../bin/ls": verify_storage.c
Q ".../bin/ls": verify_storage.o
 -> Last 288 QUOTEed lines were generated by "/bin/ls"
atlas$
```

1.13 Remote access troubleshooting

1.13.1 Server connection

When setting up `cf-serverd`, you might see the error message

```
Unspecified server refusal
```

This means that `cf-serverd` is unable or is unwilling to authenticate the connection from your client machine. The message is generic: it is deliberately non-specific so that anyone attempting to attack or exploit the service will not be given information which might be useful to them. There is a simple checklist for curing this problem:

1. Make sure that the domain variable is set in the configuration files read by both client and server; alternatively use `skipidentify` and `skipverify` to decouple DNS from the the authentication.
2. Make sure that you have granted access to your client in the server body

```
body server control
{
allowconnects         => { "127.0.0.1" , "::1" ...etc };
allowallconnects      => { "127.0.0.1" , "::1" ...etc };
trustkeysfrom         => { "127.0.0.1" , "::1" ...etc };
}
```

3. Make sure you have created valid keys for the hosts using `cf-key`.
4. If you are using secure copy, make sure that you have created a key file and that you have distributed and installed it to all participating hosts in your cluster.

Always remember that you can run CFEngine in verbose or debugging modes to see how the authentication takes place:

```
cf-agent -v
cf-serverd -v
```

`cf-agent` reports that access is denied regardless of the nature of the error, to avoid giving away information which might be used by an attacker. To find out the real reason for a denial, use verbose '`-v`' or even debugging mode '`-d2`'.

1.13.2 Key exchange

The key exchange model used by CFEngine is based on that used by OpenSSH. It is a peer to peer exchange model, not a central certificate authority model. This means that there are no scalability bottlenecks (at least by design, though you might introduce your own if you go for an overly centralized architecture).

The problem of key distribution is the conundrum of every public key infrastructure. Key exchange is handled automatically by CFEngine and all you need to do is to decide which keys to trust.

When public keys are offered to a server, they could be accepted automatically on trust because no one is available to make a decision about them. This would lead to a race to be the first to submit a key claiming identity.

Even with DNS checks for correct name/IP address correlation (turned off with `skipverify`), it might be possible to submit a false key to a server.

The server `cf-serverd` blocks the acceptance of unknown keys by default. In order to accept such a new key, the IP address of the presumed client must be listed in the `trustkeysfrom` stanza of a `server` bundle (these bundles can be placed in any file). Once a key has been accepted, it will never be replaced with a new key, thus no more trust is offered or required.

Once you have arranged for the right to connect to the server, you must decide which hosts will have access to which files. This is done with `access` rules.

```
bundle server access_rules()

{
access:

  "/path/file"

    admit   => { "127.0.0.1", "127.0.0.2", "127.0.0.3" },
```

```
    deny    => { "192\..*" };
}
```

On the client side, i.e. `cf-runagent` and `cf-agent`, there are three issues:

1. Choosing which server to connect to.
2. Trusting the identity of any previously unknown servers, i.e. trusting the server's public key to be its and no one else's. (The issues here are the same as for the server.)
3. Choosing whether data transfers should be encrypted (with `encrypt`).

Because there are two clients for connecting to `cf-serverd` (`cf-agent` and `cf-runagent`), there are also two ways of managing trust of server keys by a client. One is an automated option, setting the option `trustkey` in a `copy_from` stanza, e.g.

```
body copy_from example
     {
     # .. other settings ..
     trustkey => "true";
     }
```

Another way is to run `cf-runagent` in interactive mode. When you run `cf-runagent`, unknown server keys are offered to you interactively (as with `ssh`) for you to accept or deny manually:

```
WARNING - You do not have a public key from host ubik.iu.hio.no = 128.39.74.25
          Do you want to accept one on trust? (yes/no)
-->
```

1.13.3 Time windows (races)

Once public keys have been exchanged from client to server and from server to client, the issue of trust is solved according to public key authentication schemes. You only need to worry about trust when one side of a connection has never seen the other side before.

Often you will have a central server and many client satellites. Then the best way to transfer all the keys is to set the `trustkey` flags on server and clients sides to coincide with a time at which you know that `cf-agent` will be run, and when a spoofer is unlikely to be able to interfere.

This is a once-only task, and the chance of an attacker being able to spoof a key-transfer is small. It would require skill and inside-information about the exchange procedure, which would tend to imply that the trust model was already broken.

Another approach would be to run `cf-runagent` against all the hosts in the group from the central server and accept the keys one by one, by hand, though there is little to be gained from this.

Trusting a host for key exchange is unavoidable. There is no clever way to avoid it. Even transferring the files manually by diskette, and examining every serial number of the computers you have, the host has to trust the information you are giving it. It is all based on assertion. You can make it almost impossible for keys to be faked or attacked, but you cannot make it absolutely impossible. Security is about managing reasonable levels of risk, not about magic.

All security is based on a moment of trust, that is granted by a user at some point in time – and is assumed thereafter (once given, hard to rescind). Cryptographic key methods only remove the need

for a repeat of the trust decision. After the first exchange, trust is no longer needed, because they keys allow identity to be actually verified.

Even if you leave the trust options switched on, you are not blindly trusting the hosts you know about. The only potential insecurity lies in any new keys that you have not thought about. If you use wildcards or IP prefixes in the trust rules, then other hosts might be able to spoof their way in on trust because you have left open a hole for them to exploit. That is why it is recommended to return the system to the default state of zero trust immediately after key transfer, by commenting out the trust options.

It is possible, though somewhat laborious to transfer the keys out of band, by copying '/var/cfengine/ppkeys/localhost.pub' to /var/cfengine/ppkeys/user-aaa.bbb.ccc.mmm (assuming IPv4) on another host. e.g.

```
localhost.pub -> root-128.39.74.71.pub
```

This would be a silly way to transfer keys between nearby hosts that you control yourself, but if transferring to long distance, remote hosts it might be an easier way to manage trust.

1.13.4 Other users than root

CFEngine normally runs as user "root" (except on Windows which does not normally have a root user), i.e. a privileged administrator. If other users are to be granted access to the system, they must also generate a key and go through the same process. In addition, the users must be added to the server configuration file.

1.13.5 Encryption

CFEngine provides encryption for keeping file contents private during transfer. It is assumed that users will use this judiciously. There is nothing to be gained by encrypting the transfer of public files – overt use of encryption just contributes to global warming, burning unnecessary CPU cycles without offering any security.

The main role for encryption in configuration management is for authentication. CFEngine always uses encryption during authentication, so none of the encryption settings affect the security of authentication.

2 A simple crash course in concepts

2.1 Rules are promises

Everything in CFEngine 3 can be interpreted as a promise. Promises can be made about all kinds of different subjects, from file attributes, to the execution of commands, to access control decisions and knowledge relationships.

This simple but powerful idea allows a very practical uniformity in CFEngine syntax. There is only one grammatical form for statements in the language that you need to know and it looks generically like this:

```
type:

  classes::

   "promiser" -> { "promisee1", "promisee2", ... }

      attribute_1 => value_1,
      attribute_2 => value_2,
      ...
      attribute_n => value_n;
```

We speak of a promiser (the abstract object making the promise), the promisee is the abstract object to whom the promise is made, and them there is a list of associations that we call the 'body' of the promise, which together with the promiser-type tells us what it is all about.

Not all of these elements are necessary every time. Some promises contain a lot of implicit behaviour. In other cases we might want to be much more explicit. For example, the simplest promise looks like this:

```
commands:

  "/bin/echo hello world";
```

This promise has default attributes for everything except the 'promiser', i.e. the command string that promises to execute. A more complex promise contains many attributes:

```
files:

  "/home/mark/tmp/test_plain" -> "system blue team",

       comment => "This comment follows the rule for knowledge integration",
       perms   => users("@(usernames)"),
       create  => "true";
```

The list of promisees is not used by CFEngine except for documentation, just as the comment attribute (which can be added to any promise) has no actual function other than to provide more information to the user in error tracing and auditing.

You see several kinds of object in this example. All literal strings (e.g. `"true"`) in CFEngine 3 must be quoted. This provides absolute consistency and makes type-checking easy and error-correction powerful. All function-like objects (e.g. `users("..")`) are either builtin special functions or parameterized templates which contain the 'meat' of the right hand side.

The words `commands`, and `files` are built-in promise types. Promise types generally belong each to a particular component of CFEngine, as the components are designed to keep different kinds of promises. A few types, such as `vars`, `classes` and `reports` are common to all the different component bundles. You will find a full list of the promise types that can be made by the different components in the 'bundles' chapters that follow.

2.2 Best practice for writing promises

When writing promises, get into the habit of giving every promise a comment that explains its intention.

Also, give related promises *handles*, or labels that can be used to refer to them.

```
files:

  "/var/cfengine/inputs"

    handle => "update_policy",
    comment => "Update the configuration from a master server",

    perms => system("600"),
    copy_from => mycopy("$(master_location)","$(policy_server)"),
    depth_search => recurse("inf"),
    file_select => input_files,
    action => immediate;
```

If a promise affects another promise in some way, you can make the affected promise one of the promisees, like this:

```
access:

  "/master/cfengine/inputs" -> { "update_policy", "other_promisee" },

    comment => "Grant access to policy to our clients",
    handle  => "serve_updates",

    admit   => { "217.77.34.*" };
```

Conversely, if a promise might depend on another in some (even indirect) way, document this too.

```
files:

  "/var/cfengine/inputs"

    comment => "Update the configuration from a master server",
    handle     => "update_policy",

    depends_on => {"serve_updates"},
```

```
    perms => system("600"),
    copy_from => mycopy("$(master_location)","$(policy_server)"),
    depth_search => recurse("inf"),
    file_select => input_files,
    action => immediate;
```

Get into the habit of adding the cause-effect lines of influence. Enterprise editions of CFEngine will track the dependencies between these promises and map out impact analyses.

2.3 Containers

CFEngine allows you to group multiple promise statements into containers called bundles.

```
bundle agent identifier

{
commands:

  "/bin/echo These commands are a silly way to use CFEngine";
  "/bin/ls -l";
  "/bin/echo But they illustrate a point";

}
```

Bundles serve two purposes: they allow us to collect related promises under a single heading, like a subroutine, and they allow us to mix configuration for different parts of CFEngine in the same file. The type of a bundle is the name of the component of CFEngine for which it is intended.

For instance, we can make a self-contained example agent-server configuration by labelling the bundles:

```
#
# Not a complete example
#

bundle agent testbundle

{
files:

  "/home/mark/tmp/testcopy"

    comment      => "Throwaway example...",
    copy_from    => mycopy("/home/mark/LapTop/words","127.0.0.1"),
    perms        => system,
    depth_search => recurse("inf");

}

#

bundle server access_rules

{
```

```
access:

  "/home/mark/LapTop"

    admit   => { "127.0.0.1" };
}
```

Another type of container in CFEngine 3 is a 'body' part. Body parts exist to hide complex parameter information in reusable containers. The right hand side of some attribute assignments use body containers to reduce the amount of in-line information and preserve readability. You cannot choose where to use bodies: either they are used or they are not used for a particular kind of attribute. What you can choose, however, is the name and number of parameters for the body; and you can make as many of them as you like: For example:

```
body copy_from mycopy(from,server)

{
source      => "$(from)";
servers     => { "$(server)" };
copy_backup => "true";

special_class::

  purge       => "true";
}
```

Notice also that classes can be used in bodies as well as parameters so that you can hide environmental adaptations in these bodies also. The classes used here are effectively ANDed with the classes under which the calling promise is defined.

2.4 When and where are promises made?

When you type a promise into a CFEngine bundle, the promise will be read by every cf-agent that reads the file, each time it is called into being. For some promises this is okay, but for others you only want to verify the promise once in a while, e.g. once per day or once per hour. There are two ways to say when and where a promise applies in CFEngine:

Classes Classes are the double-colon decision syntax in CFEngine. They determine in what context a promise is made, i.e. when and where. Recall the basic syntax of a promise:

```
promise-type:

   class-expression::

      promiser -> promisee

         attribute => body,
              ifvarclass => other-class-expression;
```

The class expression may contain words like '`Hr12`', meaning from 12:00 p.m - 13:00 p.m., or '`Hr12&Min05_10`', meaning between 12:05 and 12:10. Classes may also have spatial descriptors like '`myhost`' or '`solaris`', which decide which hosts in the namespace, or '`ipv4_192_168_1_101`' which decides the location in IPv4 address space.

If the class expression is true, the promise can be considered made for the duration of the current execution.

CFEngine 3 has a new class predicate `ifvarclass` which is ANDed with the normal class expression, and which is evaluated together with the promise. It may contain variables as long as the resulting expansion is a legal class expression.

Locks Locks determine how often a promise is verified.

CFEngine is controlled by a series of locks which prevent it from checking promises too often, and which prevent it from spending too long trying to verify promises it already verified recently. The locks work in such a way that you can start several CFEngine processes simultaneously without them interfering with each other. You can control two things about each kind of action in the action sequence:

'`ifelapsed`'
: The minimum time (in minutes) which should have passed since the last time that promise was verified. It will not be executed again until this amount of time has elapsed. (Default time is 1 minute.)

'`expireafter`'
: The maximum amount (in minutes) of time cf-agent should wait for an old instantiation to finish before killing it and starting again. (Default time is 120 minutes.)

You can set these values either globally (for all actions) or for each action separately. If you set global and local values, the local values override the global ones. All times are written in units of *minutes*. Global setting is in the control body:

```
body agent control
{
ifelapsed => "60";        # one hour
}
```

or locally in the transaction bodies:

```
body action example
{
ifelapsed => "90";        # 1.5 hours
}
```

These locks do not prevent the whole of cf-agent from running, only atomic promise checks. Several different atoms can be run concurrently by different cf-agents. The locks ensure that atoms will never be started by two cf-agents at the same time, or too soon after a verification, causing contention and wasting CPU cycles.

2.5 Types in CFEngine 3

A key difference in CFEngine 3 compared to earlier versions is the presence of data types. Data types are a mechanism for associating values and checking consistency in a language. Once again, there is a simple pattern to types in CFEngine.

The principle is very simple: types exist in order to match like a plug-socket relationship. In the examples above, you can see two places where types are used to match templates:

- Matching bundles to CFEngine components (such as agent, server, common, etc.):

```
bundle TYPE name  # matches TYPE to running agent
{
}
```

- Match bodies templates to lvalues in `lvalues => rvalue` constraints:

```
body TYPE name    # matches TYPE => name in promise
{
}
```

Check these by identifying the words '`agent`' and '`copy_from`' in the examples above. Types are there to make configuration more robust.

2.6 Datatypes in CFEngine 3

CFEngine variables have two meta-types: scalars and lists. A scalar is a single value, a list is a collection of scalars. Each scalar may have one of three types: `string`, `int` or `real`. Typing is dynamic, so these are interchangable in many instances. However arguments to special functions check legal type for consistency.

Integer constants may use suffixes to represent large numbers.

- 'k' = value times 1000.
- 'K' = value times 1024.
- 'm' = value times 1000^2
- 'M' = value times 1024^2
- 'g' = value times 1000^3
- 'G' = value times 1024^3
- '%' meaning percent, in limited contexts
- 'inf' = a constant representing an unlimited value.

2.7 Variable expansion in CFEngine 3

CFEngine 3 has some simple rules for variable expansion. These make a couple of restrictions that enforce discipline of clarity and allow automatic dependency tracking in enterprise versions of CFEngine.

2.7.1 Scalar variable expansion

Scalar variables are written '`$(name)`' and they represent a single value at a time.

- Scalars that are written without a context, e.g. '`$(myvar)`' are local to the current bundle (and are equivalent to '`$(this.myvar)`').
- Scalars are globally available everywhere provided one uses the context to verify them e.g. '`$(context.myvar)`' may be written to access the variable 'myvar' in bundle 'context'.

2.7.2 List variable substitution and expansion

- Scalar references to *local* list variables imply iteration, e.g. suppose we have local list variable '`@(list)`', then the scalar '`$(list)`' implies an iteration over every value of the list.

- Lists can be passed around in their entirety in any context where a list is expected as '`@(list)`'., e.g.

```
vars:

  "longlist" slist => { @(shortlist), "plus", "plus" };

  "shortlist" slist => { "you", "me" };
```

You can pass lists to functions by parameter or by qualified reference. The first uses parameterization to map a global list into a local context.

```
#
# Show access of external lists.
#
# - to pass lists globally, use a parameter to dereference them
#

body common control
{
bundlesequence => { hardening(@(va.tmpdirs)) };
}

#########################################################

bundle common va
{
vars:

 "tmpdirs"  slist => { "/tmp", "/var/tmp", "/usr/tmp"  };

}

##########################################################

bundle agent hardening(x)
{
classes:

  "ok" expression => "any";

vars:

 "other"    slist => { "/tmp", "/var/tmp" };

reports:
```

```
  ok::

    "Do $(x)";
    "Other: $(other)";
}
```

This alternative uses a direct 'short-circuit' approach to map the global list into the local context.

```
#
# Show access of external lists.
#

body common control
{
bundlesequence => { hardening };
}

#########################################################

bundle common va
{
vars:

 "tmpdirs"  slist => { "/tmp", "/var/tmp", "/usr/tmp"  };

}

##########################################################

bundle agent hardening
{
classes:

  "ok" expression => "any";

vars:

 "other"    slist => { "/tmp", "/var/tmp" };
 "x"        slist => { @(va.tmpdirs) };

reports:

  ok::

    "Do $(x)";
    "Other: $(other)";
```

```
}
```

2.7.3 Special list value `cf_null`

As of CFEngine core version 3.1.0, the value '`cf_null`' may be used as a NULL value within lists. This value is ignored in list variable expansion.

```
vars:

  "empty_list" slist => { "cf_null" };
```

2.7.4 Arrays in CFEngine 3

Array variables are written with '`[`' and '`]`' brackets, e.g.

```
bundle agent example

{
vars:

  "component" slist => { "cf-monitord", "cf-serverd", "cf-execd" };

  "array[cf-monitord]" string => "The monitor";
  "array[cf-serverd]" string => "The server";
  "array[cf-execd]" string => "The executor, not executioner";

commands:

   "/bin/echo $(component) is"

            args => "$(array[$(component)])";

}
```

Arrays are associative and may be of type scalar or list. Enumerated arrays are simply treated as a special case of associative arrays, since there are no numerical loops in CFEngine. Special functions exist to extract lists of keys from array variables for iteration purposes.

Thus one could have written the example above in the form of the following example. Note, too, that the use of `getindices` avoids the earlier poor practice of repeating the enumeration of key names, and instead uses the better strategy of automatically deriving them.

```
bundle agent example

{
vars:

  "array[cf-monitord]" string => "The monitor";
```

```
  "array[cf-serverd]" string => "The server";
  "array[cf-exced]" string => "The executor, not executioner";

  "component" slist => getindices("array");

commands:

  "/bin/echo $(component) is"

           args => "$(array[$(component)])";
}
```

2.8 Name spaces

Name spaces are private bundle and body contexts, allowing multiple files to define the bundles and bodies with the same name, without conflict.

To isolate a file into its own name space, you add a control promise to the file before the relevant bundles or bodies. All files start off in the default namespace if you don't explicitly set this. Once set, this applies until the end of the file or the next namespace change.

```
body file control
{
namespace => "myspace";
}
```

To distinguish the bundle `mymethod` in the default namespace from one in another namespace, you prefix the bundle name with the namespace, separated by a colon.

```
methods:

  "namespace demo" usebundle => myspace:mymethod("arg1");

  "namespace demo" usebundle => mymethod("arg1","arg2");
```

To distinguish a body from one in another namespace, you can prefix the body name with the namespace, separated by a colon.

```
files:

   "/file"

       create => "true",
        perms => name1:settings;
```

The default namespace, i.e. that which is implied by not making any namespace declarations, can be accessed or referred to by prefixing with the default string

```
files:

   "/file"
```

```
        create => "true",
         perms => default:settings;
```

For example, this can be used to refer to standard library objects from within a private namespace.

Global classes are not handled by namespaces, and you are advised to prefix them with the namespace like this:

```
files:

   "/file"

        create => "true",
        action => if_repaired("namespace_done");
```

This is not prepended automatically because references to this class in class expressions cannot be detected and modified automatically.

To access variables or meta-data in bundles in a different namespace, use the colon as a namespace prefix:

```
 $(namespace:bundle.variable)
 $(namespace:bundle_meta.variable)
```

2.9 Normal ordering

CFEngine takes a pragmatic point of view to ordering. When promising 'scalar' attributes and properties, ordering is irrelevant and need not be considered. More complex patterned data structures require ordering to be preserved, e.g. editing in files. CFEngine solves this in a two-part strategy:

- CFEngine maintains a default order of promise-types. This is based on a simple logic of what needs to come first, e.g. it makes no sense to create something and then delete it, but it could make sense to delete and then create (an equilibrium). This is called *normal ordering* and is described below.
- You can override normal ordering in exceptional circumstances by making a promise in a class context and defining that class based on the outcome of another promise.

2.9.1 Agent normal ordering

1. CFEngine tries to keep variable and class promises before starting to consider any other kind of promise. In this way, global variable and classes can be set, as well as creating `classes` promises, upon which later agent-bundle `vars` promises may depend. Place these at the start of your configuration (see next item).
2. If you set variables based on classes that are determined by variables, in a complex dependency chain, then you introduce an order dependence to the resolution that might be non-unique. Since CFEngine starts trying to converge values as soon as possible, it is best to define variables in bundles before using them, i.e. as early as possible in your configuration. In general it is wise to avoid class-variable dependency as much as possible.

3. CFEngine executes agent promise bundles in the strict order defined by the `bundlesequence` (possibly overridden by the `-b` or `--bundlesequence` command line option).
4. Within a bundle, the promise types are executed in a round-robin fashion according to so-called 'normal ordering' (essentially deletion first, followed by creation). The actual sequence continues for up to three iterations of the following, converging towards a final state:

   ```
   vars
   classes
   outputs
   interfaces
   files
   packages
   guest_environments
   methods
   processes
   services
   commands
   storage
   databases
   reports
   ```

   Within `edit_line` bundles in `files` promises (See 'File editing in CFEngine 3' for important details), the normal ordering is:

   ```
   vars
   classes
   delete_lines
   field_edits
   insert_lines
   replace_patterns
   reports
   ```

5. The order of promises within one of the above types follows their top-down ordering within the bundle itself
6. The order may be overridden by making a promise depend on a class that is set by another promise.

2.9.2 Server normal ordering

As with the agent, common bundles are executed before any server bundles; following this *all* server bundles are executed (the `bundlesequence` is only used for `cf-agent`). Within a server bundle, the promise types are unamgibuous. Variables and classes are resolved in the same way as the agent. On connection, access control must be handled first, then a role request might be made once access has been granted. Thus ordering is fully constrained by process with no additional freedoms.

Within a server bundle, the normal ordering is:

```
vars
classes
access
roles
```

2.9.3 Monitor normal ordering

As with the agent, common bundles are executed before any monitor bundles; following this *all* monitor bundles are executed (the `bundlesequence` is only used for `cf-agent`). Variables and classes are resolved in the same way as the agent.

Within a monitor bundle, the normal ordering is:

```
vars
classes
measurements
reports
```

2.9.4 Knowledge normal ordering

As with the agent, common bundles are executed before any knowledge bundles; following this *all* knowledge bundles are executed (the `bundlesequence` is only used for `cf-agent`). Variables and classes are resolved in the same way as the agent.

Within a knowledge bundle, the normal ordering is:

```
vars
classes
topics
occurrences
inferences
reports
```

2.10 Loops and lists in CFEngine 3

There are no explicit loops in CFEngine, instead there are lists. To make a loop, you simply refer to a list as a scalar and CFEngine will assume a loop over all items in the list.

For example, in the examples below the list `component` has three elements. The list as a whole may be referred to as `@(component)`, in order to pass the whole list to a promise where a list is expected. However, if we write `$(component)`, i.e. the scalar variable, then CFEngine assumes that it should substitute each scalar from the list in turn, and thus iterate over the list elements using a loop.

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "component" slist => { "cf-monitord", "cf-serverd", "cf-execd" };
```

```
  "new_list" slist => { "cf-know", @(component) };

processes:

  "$(component)" restart_class => canonify("start_$(component)");

commands:

   "/bin/echo /var/cfengine/bin/$(component)"

       ifvarclass => canonify("start_$(component)");
}
```

If a variable is repeated, its value is tied throughout the expression; so the output of:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "component" slist => { "cf-monitord", "cf-serverd", "cf-execd" };

  "array[cf-monitord]" string => "The monitor";
  "array[cf-serverd]" string => "The server";
  "array[cf-execd]" string => "The executor, not executioner";

commands:

   "/bin/echo $(component) is"

            args => "$(array[$(component)])";
}
```

is as follows:

```
Q ".../bin/echo cf-mo": cf-monitord is The monitor
 -> Last 1 QUOTEed lines were generated by "/bin/echo cf-monitord is The monitor"
Q ".../bin/echo cf-se": cf-serverd is The server
```

```
 -> Last 1 QUOTEed lines were generated by "/bin/echo cf-serverd is The server"
Q ".../bin/echo cf-ex": cf-execd is The executor, not executioner
 -> Last 1 QUOTEed lines were generated by "/bin/echo cf-execd is The executor, not executioner"
```

2.11 Pattern matching and referencing

One of the strengths of CFEngine 3 is the ability to recognize and exploit patterns. All string patterns in CFEngine 3 are matched using PCRE regular expressions.

CFEngine has the ability to extract back-references from pattern matches. This makes sense in two cases. Back references are fragments of a string that match parenethetic expressions. For instance, suppose we have the string:

```
Mary had a little lamb ...
```

and apply the regular expression

```
"Mary ([^l]+)little (.*)"
```

The pattern matches the entire string, and it contains two parenthesized subexpressions, which respectively match the fragments 'had a ' and 'lamb ...'. The regular expression libraries assign *three* matches to this result, labelled 0, 1 and 2.

The zeroth value is the entire string matched by the total expression. The first value is the fragment matched by the first parenthesis, and so on.

Each time CFEngine matches a string, these values are assigned to a special variable context `$(match.n)`. The fragments can be referred to in the remainder of the promise. There are two places where this makes sense. One is in pattern replacement during file editing, and the other is in searching for files.

Consider the examples below:

```
bundle agent testbundle

{
files:

  # This might be a dangerous pattern - see explanation in the next section
  # on "Runaway change warning"

  "/home/mark/tmp/cf([23])?_(.*)"
       edit_line => myedit("second backref: $(match.2)");
}
```

There are other filenames that could match this pattern, but if, for example, there were to exist a file '`/home/mark/tmp/cf3_test`', then we would have:

'`$(match.0)`'
    equal to '/home/mark/tmp/cf3_test'

'$(match.1)'
: equal to '3'

'$(match.2)'
: equal to 'test'

Note that because the pattern allows for an *optional* '2' or '3' to follow the letters 'cf', it is possible that $(match.1) would contain the empty string. For example, if there was a file named '/home/mark/tmp/cf_widgets', then we would have

'$(match.0)'
: equal to '/home/mark/tmp/cf_widgets'

'$(match.1)'
: equal to ''

'$(match.2)'
: equal to 'widgets'

Now look at the edit bundle. This takes a parameter (which is the back-reference from the filename match), but it also uses back references to replace shell comment lines with C comment lines (the same approach is used to hash-comment lines in files). The back-reference variables $(match.*n*) refer to the most recent pattern match, and so in the 'C_comment' body, they do not refer to the filename components, but instead to the hash-commented line in the 'replace_patterns' promise.

```
bundle edit_line myedit(parameter)
  {
  vars:

   "edit_variable" string => "private edit variable is $(parameter)";

  insert_lines:

     "$(edit_variable)";

  replace_patterns:

  # replace shell comments with C comments

   "#(.*)"

      replace_with => C_comment,
     select_region => MySection("New section");

  }

########################################
# Bodies
########################################
```

```
body replace_with C_comment

{
replace_value => "/* $(match.1) */"; # backreference from replace_patterns
occurrences => "all";  # first, last, or all
}

########################################################

body select_region MySection(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}
```

Try this example on the file

```
[First section]

one
two
three

[New section]

four
#five
six

[final]

seven
eleven
```

The resulting file is edited like this:

```
[First section]

one
two
three

[New section]

four
/* five */
six
```

```
[final]

seven
eleven

private edit variable is second backref: test
```

2.11.1 Runaway change warning

Be careful when using patterns to search for files that are altered by CFEngine if you are not using a file repository. Each time CFEngine makes a change it saves an old file into a copy like 'cf3_test.cf-before-edit'. These new files then get matched by the same expression above – because it ends in the generic.*), or does not specify a tail for the expression. Thus CFEngine will happily edit backups of the edit file too, and generate a recursive process, resulting in something like the following:

```
cf3_test                  cf3_test.cf-before-edit
cf3_test~                 cf3_test~.cf-before-edit.cf-before-edit
cf3_test~.cf-before-edit  cf3_test~.cf-before-edit.cf-before-edit.cf-before-edit
```

Always try to be as specific as possible when specifying patterns. A lazy approach will often come back to haunt you.

2.11.2 Commenting lines

The following example shows how you would hash-comment lines in a file using CFEngine 3.

```
######################################################################
#
# HashCommentLines implemented in CFEngine 3
#
######################################################################

body common control

{
version => "1.2.3";
bundlesequence  => { "testbundle"  };
}

########################################################

bundle agent testbundle

{
files:

  "/home/mark/tmp/comment_test"

       create    => "true",
       edit_line => comment_lines_matching;
}
```

```
########################################################

bundle edit_line comment_lines_matching
  {
  vars:

    "regexes" slist => { "one.*", "two.*", "four.*" };

  replace_patterns:

   "^($(regexes))$"
      replace_with => comment("# ");
  }

########################################
# Bodies
########################################

body replace_with comment(c)

{
replace_value => "$(c) $(match.1)";
occurrences => "all";
}
```

2.11.3 Regular expressions in paths

When applying regular expressions in paths, the path will first be split at the path separators, and each element matched independently. For example, this makes it possible to write expressions like `"/home/.*/file"` to match a single file inside a lot of directories — the .* does not eat the whole string.

Note that whenever regular expressions are used in paths, the / is always used as the path separator, even on Windows. However, on Windows, if the pathname is interpreted literally (no regular expressions), then the backslash is also recognized as the path separator. This is because the backslash has a special (and potentially ambiguous) meaning in regular expressions (a `\d` means the same as `[0-9]`, but on Windows it could also be a path separator and a directory named `d`).

The `pathtype` attribute allows you to force a specific behavior when interpreting pathnames. By default, CFEngine looks at your pathname and makes an educated guess as to whether your pathname contains a regular expression. The values `"literal"` and `"regex"` explicitly force CFEngine to interpret the pathname either one way or another.

(see the `pathtype` attribute).

```
body common control
{
bundlesequence => { "wintest" };
```

```
}

#########################################

bundle agent wintest
{
files:
  "c:/tmp/file/f.*"                 # "best guess" interpretation
    delete => nodir;


  "c:\tmp\file"
    delete => nodir,
    pathtype => "literal";# force literal string interpretation


  "C:/windows/tmp/f\d"
    delete => nodir,
    pathtype => "regex";  # force regular expression interpretation
}

#########################################

body delete nodir
{
rmdirs => "false";
}
```

Note that the path '/tmp/gar.*' will only match filenames like '/tmp/gar', '/tmp/garbage' and '/tmp/garden'. It will *not* match filename like '/tmp/gar/baz' (because even though the '.*' in a regular expression means "zero or more of any character", CFEngine restricts that to mean "zero or more of any character *in a path component*"). Correspondingly, CFEngine also restricts where you can use the '/' character (you can't use it in a character class like '[^/]' or in a parenthesized or repeated regular expression component.

This means that regular expressions which include "optional directory components" won't work. You can't have a files promise to tidy the directory '(/usr)?/tmp'. Instead, you need to be more verbose and specify '/usr/tmp|/tmp', or even better, think declaratively and create an *slist* that contains both the strings '/tmp' and '/usr/tmp', and then allow CFEngine to iterate over the list!

This also means that the path '/tmp/.*/something' will match files like '/tmp/abc/something' or '/tmp/xyzzy/something'. However, even though the pattern '.*' means "zero or more of any character (except '/')", CFEngine matches files bounded by directory separators. So even though the pathname '/tmp//something' is technically the same as the pathname '/tmp/something', the regular expression '/tmp/.*/something' will *not* match on the degenerate case of '/tmp//something' (or '/tmp/something').

2.11.4 Anchored vs. unanchored regular expressions

CFEngine uses the full power of regular expressions, but there are two "flavors" of regex. Because they behave somewhat differently (while still utilizing the same syntax), it is important to know which one is used for a particular component of CFEngine:

- An "anchored" regular expression will only successfully match an entire string, from start to end. An anchored regular expression behaves as if it starts with '`^`' and ends with '`$`', whether you specify them yourself or not. Furthermore, an anchored regular expression cannot have these automatic anchors removed.
- An "unanchored" regular expression may successfully match anywhere in a string. An unanchored regex may use anchors (such as '`^`', '`$`', '`\A`', '`\Z`', '`\b`', etc.) to restrict where in the string it may match. That is, an unanchored regular expression may be easily converted into a partially- or fully-anchored regex.

For example, the `comment` parameter in `readstringarray` is an unanchored regex (see Section 11.74 [Function readstringarray], page 477). If you specify the regular expression as `"#.*"`, then on any line which contains a pound sign, everything from there until the end of the line will be removed as a comment. However, if you specify the regular expression as `"^#.*"` (note the '`^`' anchor at the start of the regex), then only lines which *start* with a '`#`' will be removed as a comment! If you want to ignore C-style comment in a multi-line string, then you have to a bit more clever, and use this regex: `"(?s)/*.*?*/"`

Conversely, `delete_lines` promises use anchored regular expressions to delete lines. If our promise uses `"bob:\d*` as a line-matching regex, then only the second line of this file will be deleted (because only the second line starts with '`bob:`' and is then followed exclusively by digits, all the way to the end of the string).

```
bobs:your:uncle
bob:111770
thingamabob:1234
robert:bob:xyz
i:am:not:bob
```

If CFEngine expects an unanchored regular expression, then finding every line that contains the letters '`bob`' is easy. You just use the regex `"bob"`. But if CFEngine expects an anchored regular expression, then you must use `".*bob.*"`.

If you want to find every line that has a field which is exactly '`bob`' with no characters before or after, then it is only a little more complicated if CFEngine expects an unanchored regex: `"(^|:)bob(:|$)"`. But if CFEngine expects an anchored regular expression, then it starts getting ugly, and you'd need to use `"bob:.*|.*:bob:.*|.*:bob"`.

2.11.5 Special topics on Regular Expressions

Regular expressions are a complicated subject, and really are beyond the scope of this document. However, it is worth mentioning a couple of special topics that you might want to know of when using regular expressions.

The first is how to *not* get a backreference. If you want to have a parenthesized expression that does not generate a back reference, there is a special PCRE syntax to use. Instead of using `()` to bracket the piece of a regular expression, use `(?:)` instead. For example, this will match the filenames '`foolish`', '`foolishly`', '`bearish`', '`bearishly`', '`garish`', and '`garishly`' in the '`/tmp`' directory. The variable

$match.0 will contain the full filename, and $match.1 will either contain the string ‘ly’ or the empty string. But the (?:expression) which matches foo, bear, or gar does *not* create a back-reference:

```
files:
    "/tmp/(?:foo|bear|gar)ish(ly)?"
```

Note that sometimes multi-line strings are subject to be matched by regular expressions. CFEngine internally matches all regular expressions using PCRE_DOTALL option, so . matches newlines. If you want to match any character except newline you could use \N escape sequence.

Another thing you might want to do is ignore capitalization. CFEngine is case-sensitive (in all things), so the files promise ‘/tmp/foolish’ will not match the files ‘/tmp/Foolish’ or ‘/tmp/fOoLish’, etc. There are two ways to acheive case-insensitivity. The first is to use character classes:

```
files:
    "/tmp/[Ff][Oo][Oo][Ll][Ii][Ss][Hh]"
```

While this is certainly correct, it can also lead to unreadability. The PCRE patterns in CFEngine have another way of introducing case-insensitvity into a pattern:

```
files:
    "/tmp/(?i:foolish)"
```

The (?i:) brackets impose case-insensitive matching on the text that it surrounds, without creating a sub-expression. You could also write the regular expression like this (but be aware that the two expressions are different, and work slightly differently, so check the documentation for the specifics):

```
files:
    "/tmp/(?i)foolish"
```

The /s, /m, and /x switches from PCRE are also available, but use them with great care!

2.12 Distributed discovery

CFEngine’s philosophy and modus operandi is to make machines as self-reliant as possible. This is the path to scalability. Sometimes we want machines to be able to detect one another and sample each others’ behaviour. This can be accomplished using probes and server functions.

For example, testing whether services are up and running can be a useful probe even from a local host. CFEngine has in-built functions for generically probing the environment; these are designed to encourage decentralized monitoring.

```
body common control

{
bundlesequence  => { "test"  };
}

###########################################################

bundle agent test

{
```

```
vars:

 "hosts" slist => { "server1.example.org", "server2", "server3" };

 "up_servers" int =>  selectservers("@(hosts)","80","","","100","alive_servers");

classes:

  "someone_alive" expression => isgreaterthan("$(up_servers)","0");

  "i_am_a_server" expression => regarray("up_servers","$(host)|$(fqhost)");

reports:

  someone_alive::

    "Number of active servers $(up_servers)" action => always;

    "First server $(alive_servers[0]) fails over to $(alive_servers[1])";


}
```

3 How to run CFEngine 3 examples

The CFEngine 'tests' directory contains a multitude of examples of CFEngine 3 code. These instructions assume that you have all of your configuration in a single test file, such as the example in the distribution directory 'tests/units'.

1. Test the file as a non-privileged user first, if you can.
2. Always verify syntax first with cf-promises. This requires no privileges. An cf-agent will not execute a configuration that has not passed this test.

```
host$ cf-promises -f ./inputfile.cf
```

3. Run the examples like this, e.g.

```
host$ src/cf-promises -f ./tests/units/unit_server_copy_localhost.cf
host$ src/cf-serverd -f ./tests/units/unit_server_copy_localhost.cf
host$ src/cf-agent -f ./tests/units/unit_server_copy_localhost.cf
```

Running cf-agent in verbose mode provides detailed information about the state of the systems promises.

```
Outcome of version 1.2.3: Promises observed to be kept 99%,
Promises repaired 1%, Promises not repaired 0%
```

The log-file 'WORKDIR/promise.log' contains the summary of these reports with timestamps. This is the simplest kind of high level audit record of the system.

4 A complete configuration

To illustrate a complete configuration for agents and daemons, consider the following example code, supplied in the 'inputs/' directory of the distribution. Comments indicate the thinking behind this starting point.

4.1 'promises.cf'

This file is the first file that cf-agent with no arguments will try to look for. It should contain all of the basic configuration settings, including a list of other files to include. In normal operation, it must have a bundlesequence.

This file can stay fixed, except for extending the bundlesequence. The bundlesequence acts like the 'genetic makeup' of the configuration. In a large configuration, you might want to have a different bundlesequence for different classes of host, so that you can build a complete system like a check-list from different combinations of building blocks. You can construct different lists by composing them from other lists, or you can use methods promises as an alternative for composing bundles for different classes.

```
#######################################################
#
# promises.cf
#
#######################################################

body common control

{
# List the 'genes' for this system..

bundlesequence  => {
                   "update",
                   "garbage_collection",
                   "main",
                   "cfengine"
                   };


inputs          => {
                   "update.cf",
                   "site.cf",
                   "library.cf"
                   };
}

#######################################################
# Now set defaults for all components' hard-promises
#######################################################
```

```
body agent control
{
# if default runtime is 5 mins, we need more for long jobs
ifelapsed => "15";
}

#######################################################

body monitor control
{
forgetrate => "0.7";
}

##########si############################################

body executor control

{
splaytime => "1";
mailto => "cfengine_mail@example.org";
smtpserver => "localhost";
mailmaxlines => "30";

# Instead of a separate update script, now do this

exec_command => "$(sys.workdir)/bin/cf-agent -f failsafe.cf && $(sys.workdir)/bin/cf-agent";
}

#######################################################

body reporter control

{
reports => { "performance", "last_seen", "monitor_history" };
build_directory => "/tmp/nerves";
report_output => "html";
}

#######################################################

body runagent control
{
hosts => {
          "127.0.0.1"
          # , "myhost.example.com:5308", ...
         };
```

```
}

#########################################################

body server control

{
allowconnects         => { "127.0.0.1" , "::1" };
allowallconnects      => { "127.0.0.1" , "::1" };
trustkeysfrom         => { "127.0.0.1" , "::1" };

# Make updates and runs happen in one

cfruncommand => "$(sys.workdir)/bin/cf-agent";

allowusers   => { "root" };
}
```

4.2 'site.cf'

Use this file to add your site-specific configuration. Common bundles can be used to define global variables. Otherwise, unqualified variables are local to the bundle in which they are defined – however they can be accessed by writing `$(bundle_name.variable_name)`.

```
#########################################################
#
# site.cf
#
#########################################################

bundle common g
{
vars:

  SuSE::

   "crontab" string => "/var/spool/cron/tabs/root";

 !SuSE::

   "crontab" string => "/var/spool/cron/crontabs/root";
}
```

The CFEngine bundle below detects whether CFEngine 2 is already running on the host or not, and if so attempts to kill off old daemon processes and encapsulate the agent. It also looks for rules in the old CFEngine configuration that would potentially spoil CFEngine 3's control of the system: the last thing we want is for CFEngine 2 and CFEngine 3 to fight each other for control of the system.

CFEngine 3 tries to edit an existing crontab entry to replace any references to `cfexecd` with `cf-execd`; if none are found it will add a 5 minute run schedule. You should never put `cf-agentor` `cf-agent` directly inside `cron` without the `cf-execd` wrapper.

```
#######################################################
# Start with CFEngine itself
#######################################################

bundle agent cfengine

{
classes:

  "integrate_cfengine2"

      and => {
             fileexists("$(sys.workdir)/inputs/cfagent.conf"),
             fileexists("$(sys.workdir)/bin/cfagent")
             };

vars:

   "cf2bits" slist => { "cfenvd", "cfservd", "cfexecd" };

commands:

 integrate_cfengine2::

   "$(sys.workdir)/bin/cfagent"

        action => longjob;

files:

  # Warn about rules relating to CFEngine 2 in inputs - could conflict

  "$(sys.workdir)/inputs/.*"

       comment     => "Check if there are still promises about CFEngine 2 that need removing",
       edit_line   => DeleteLinesMatching(".*$(cf2bits).*"),
       file_select => OldCf2Files,
       action      => WarnOnly;

  # Check cf-execd and schedule is in crontab

  "$(g.crontab)"
       edit_line => upgrade_cfexecd,
```

```
         classes => define("exec_fix");

processes:

  exec_fix::

    "cron" signals => { "hup" };


}

#######################################################
# General site issues can be in bundles like this one
#######################################################

bundle agent main

{
vars:

  "component" slist => { "cf-monitord", "cf-serverd" };

 # - - - - - - - - - - - - - - - - - - - - - - - - -

files:

  "$(sys.resolv)"  # test on "/tmp/resolv.conf" #

     create        => "true",
     edit_line     => resolver,
     edit_defaults => def;

 # Uncomment this to perform a change-detection scan

 #  "/usr"
 #     changes      => lay_trip_wire,
 #     depth_search => recurse("inf"),
 #     action       => measure;

processes:

  "cfenvd"             signals => { "term" };

 # Uncomment this when you are ready to upgrade the server
 #
 #  "cfservd"             signals => { "term" };
 #
```

```
 # Now make sure the new parts are running, cf-serverd will fail if
 # the old server is still running

  "$(component)" restart_class => canonify("start_$(component)");

 # - - - - - - - - - - - - - - - - - - - - - - - - -

commands:

   "$(sys.workdir)/bin/$(component)"

       ifvarclass => canonify("start_$(component)");

}
```

This section takes a backup of a user home directory. This is especially useful for a single laptop or personal workstation that does not have a regular external backup. If a user deletes a file by accident, this shadow backup might contain the file even while travelling offline.

```
#######################################################
# Backup
#######################################################

bundle agent backup
{
files:

  "/home/backup"

     copy_from => cp("/home/mark"),
  depth_search => recurse("inf"),
   file_select => exclude_files,
        action => longjob;

}

#######################################################
# Garbage collection issues
#######################################################

bundle agent garbage_collection
{
files:

  "$(sys.workdir)/outputs"
```

```
    delete => tidy,
    file_select => days_old("3"),
    depth_search => recurse("inf");


}

###########################################################

body file_select OldCf2Files
{
leaf_name => {
             "promises\.cf",
             "site\.cf",
             "library\.cf",
             "failsafe\.cf",
             ".*\.txt",
             ".*\.html",
             ".*~",
             "#.*"
             };

file_result => "!leaf_name";
}

###########################################################

body action measure
{
measurement_class => "Detect Changes in /usr";
ifelapsed => "240";       # 4 hours
expireafter => "240";     # 4 hours
}
```

Some basic anomaly detection: we respond with simple warnings if resource anomalies are detected.

```
#######################################################
# Anomaly monitoring
#######################################################

bundle agent anomalies
{
reports:

rootprocs_high_dev2::
```

```
   "RootProc anomaly high 2 dev on $(mon.host) at $(mon.env_time)
    measured value $(mon.value_rootprocs) av $(mon.av_rootprocs)
    pm $(mon.dev_rootprocs)"

      showstate => { "rootprocs" };

entropy_www_in_high&anomaly_hosts.www_in_high_anomaly::

   "HIGH ENTROPY Incoming www anomaly high anomaly dev!!
    on $(mon.host) at $(mon.env_time)
    - measured value $(mon.value_www_in)
    av $(mon.av_www_in) pm $(mon.dev_www_in)"

      showstate => { "incoming.www" };

 entropy_www_in_low.anomaly_hosts.www_in_high_anomaly::

   "LOW ENTROPY Incoming www anomaly high anomaly dev!!
    on $(mon.host) at $(mon.env_time)
     - measured value $(svalue_www_in)
    av $(av_www_in) pm $(dev_www_in)"

     showstate => { "incoming.www" };

entropy_tcpsyn_in_low.anomaly_hosts.tcpsyn_in_high_dev2::

   "Anomalous number of new TCP connections on $(mon.host)
    at $(mon.env_time)
    - measured value $(mon.value_tcpsyn_in)
    av $(mon.av_tcpsyn_in) pm $(mon.dev_tcpsyn_in)"

     showstate => { "incoming.tcpsyn" };

 entropy_dns_in_low.anomaly_hosts.dns_in_high_anomaly::

   "Anomalous (3dev) incoming DNS packets on $(mon.host)
    at $(mon.env_time) - measured value $(mon.value_dns_in)
    av $(av_dns_in) pm $(mon.dev_dns_in)"

     showstate => { "incoming.dns" };

 entropy_dns_in_low.anomaly_hosts.udp_in_high_dev2::

   "Anomalous (2dev) incoming (non-DNS) UDP traffic
    on $(mon.host) at $(mon.env_time) - measured value
    $(mon.value_udp_in) av $(mon.av_udp_in) pm $(mon.dev_udp_in)"
```

```
     showstate => { "incoming.udp" };

 anomaly_hosts.icmp_in_high_anomaly.!entropy_icmp_in_high::

   "Anomalous low entropy (3dev) incoming ICMP traffic
    on $(mon.host) at $(mon.env_time) - measured value $(mon.value_icmp_in)
    av $(mon.av_icmp_in) pm $(mon.dev_icmp_in)"

     showstate => { "incoming.icmp" };
}
```

Server access rules are a touchy business. In an enterprise setting you generally want every host to allow a monitoring host to be able to download data, and a backup host to be able to access important data on every host. On a laptop or personal workstation, there might not be any reason to run a server for external use; however you might configure it as below to allow localhost access for testing.

```
#######################################################
# Server configuration
#######################################################

bundle server access_rules()
{
access:

  "/home/mark/test_area"

    admit   => { "127.0.0.1" };

  # Rule for cf-runagent

  "/home/mark/.cfagent/bin/cf-agent"

    admit   => { "127.0.0.1" };

# New in cf3 - RBAC with cf-runagent

roles:

  ".*"  authorize => { "mark" };
}
```

4.3 ‘update.cf’

This file should rarely if ever change. Should you ever change it (or when you upgrade CFEngine), take special care to ensure the old and the new CFEngine can parse and execute this file successfully. If

not, you risk losing control of your system (that is, if CFEngine cannot successfully execute this set of promises, it has no mechanism for distributing *new* policy files).

By default, the policy defined in 'update.cf' is executed from two sets of promise bodies. The "usual" one (defined in the bundlesequence in 'promises.cf') and another in the backup/failsafe bundlesequence (defined in 'failsafe.cf').

```
#########################################################
#
# update.cf
#
#########################################################

bundle agent update
{
vars:

 "master_location" string => "/your/master/cfengine-inputs";

files:

  # Update the configuration

  "/var/cfengine/inputs"

    perms => system("600"),
    copy_from => mycopy("$(master_location)","localhost"),
    depth_search => recurse("inf"),
    action => immediate;

}

############################################

body perms system(p)

{
mode  => "$(p)";
}

############################################

body file_select cf3_files

{
leaf_name => { "cf-.*" };

file_result => "leaf_name";
```

```
}

#########################################################

body copy_from mycopy(from,server)

{
source      => "$(from)";
compare     => "digest";
}

#########################################################

body action immediate
{
ifelapsed => "1";
}
```

4.4 'failsafe.cf'

This file should probably never change. The only job of '`failsafe.cf`' is to execute the `update` bundle in a "standalone" context should there be a syntax error somewhere in the main set of promises. In this way, if a client machine's policies are ever corrupted after downloading erroneous policy from a server, that client will have a failsafe method for downloading a corrected policy once it becomes available on the server. Note that by "corrupted" and "erroneous" we typically mean "broken via administrator error" - mistakes happen, and the '`failsafe.cf`' file is CFEngine's way of being prepared for that eventuality.

If you ever change '`failsafe.cf`' (or when you upgrade CFEngine), make sure the old and the new CFEngine can successfully parse and execute this file. If not, you risk losing control of your system (that is, if CFEngine cannot successfully execute this policy file, it has no failsafe/fallback mechanism for distributing *new* policy files).

```
#########################################################
#
# Failsafe file
#
#########################################################

body common control

{
bundlesequence => { "update" };

inputs => { "update.cf" };
}

############################################
```

```
body depth_search recurse(d)

{
depth => "$(d)";
}
```

4.5 What should a failsafe and update file contain?

The 'failsafe.cf' file is to make sure that your system can upgrade gracefully to new versions even when mistakes are made.

As a general rule:

- Upgrade the software first, then add new features to the configuration.
- Never use advanced features in the failsafe or update file.
- Avoid using library code (including any bodies from 'cfengine_stdlib.cf'). Copy/paste any bodies you need using a unique name that does not collide with a name in library (we recommend simply adding the prefix "u_"). This may mean that you create duplicate functionality, but that is okay in this case to ensure a 100% functioning *standalone* update process). The promises which manage the update process should not have *any* dependencies on any other files.

A CFEngine configuration will fail-over to the failsafe.cf configuration if it is unable to read or parse the contents successfully. That means that any syntax errors you introduce (or any new features you utilize in a configuration) will cause a fail-over, because the parser will not be able to interpret the policy. If the failover is due to the use of new features, they will not parse until the software itself has been updated (so we recommend that you always update CFEngine before updating policy to use new features). If you accidentally cause a bad (i.e., unparseable) policy to be distributed to client machines, the failsafe.cf policy on those machines will run (and will eventually download a working policy, once you fix it on the policy host).

4.6 Recovery from errors in the configuration

The 'failsafe.cf' file should be able to download the latest master configuration from source always.

```
#######################################################
#
# failsafe.cf
#
#######################################################

body common control

{
bundlesequence => { "update" };
}

#########################################################

bundle agent update
```

```
{
files:

  "/var/cfengine/inputs"

    perms => system,
    copy_from => mycopy("/home/mark/cfengine-inputs","localhost"),
    file_select => cf3_files,
    depth_search => recurse("inf");
}

#########################################################

body perms system

{
mode  => "0700";
}

#########################################################

body depth_search recurse(d)

{
depth => "$(d)";
}

############################################

body file_select cf3_files

{
leaf_name => { "cf-.*" };

file_result => "leaf_name";
}

#########################################################

body copy_from mycopy(from,server)

{
source       => "$(from)";
servers      => { "$(server)" , "failover.domain.tld" };
#copy_backup => "true";
#trustkey    => "true";
encrypt      => "true";
```

```
}
```

If the `copy_backup` option is true, CFEngine will keep a single previous version of the file before copy, if the value is '`timestamp`' CFEngine keeps time-stamped versions either in the location of the file, or in the file repository if one is defined. The `trustkey` option should normally be commented out so that public keys are only exchanged under controlled conditions.

4.7 Recovery from errors in the software

The update should optionally include an update of software so that a single failover from a configuration that is 'too new' for the software will still correct itself once the new software is available.

```
#######################################################
#
# update.cf
#
#######################################################

bundle agent update

{
files:

  "/var/cfengine/inputs"

    perms => system("600"),
    copy_from => mycopy("/home/mark/cfengine-inputs","localhost"),
    depth_search => recurse("inf");
}

############################################

body perms system(p)

{
mode  => "$(p)";
}

############################################

body file_select cf3_files

{
leaf_name => { "cf-.*" };

file_result => "leaf_name";
}
```

```
#########################################################

body copy_from mycopy(from,server)

{
source      => "$(from)";
compare     => "digest";
}
```

5 Control promises

While promises to configure your system are entirley user-defined, the details of the operational behaviour of the CFEngine software is of course hard-coded. You can still configure the details of this behaviour using the control promise bodies. Control behaviour is defined in bodies because the actual promises are fixed and you only change their details within sensible limits.

Note that in CFEngine's previous versions, the `control` part of the configuration contained a mixture of internal control parameters and user definitions. There is now a cleaner separation in CFEngine 3. User defined behaviour requires a promise, and must therefore be defined in a bundle.

Below is a list of the control parameters for the different components (Agents and Daemons[1]) of the CFEngine software.

5.1 `common` control promises

```
body common control

{
inputs  => {
           "update.cf",
           "library.cf"
           };

bundlesequence  => {
                   update("policy_host.domain.tld"),
                   "main",
                   "cfengine2"
                   };

goal_categories => { "goals", "targets", "milestones" };
goal_patterns   => { "goal_.*", "target.*" };

output_prefix => "cfengine>";
version => "1.2.3";
}
```

The `common` control body refers to those promises that are hard-coded into all the components of CFEngine, and therefore affect the behaviour of all the components.

5.1.1 `bundlesequence`

Type: slist

Allowed input range: `.*`

[1] There is no Da Vinci code in CFEngine

Synopsis: List of promise bundles to verify in order

Example:

```
body common control

{
bundlesequence  => {
                   update("policy_host.domain.tld"),
                   "main",
                   "cfengine2"
                   };
}
```

Notes:

The `bundlesequence` determines which of the compiled bundles will be executed and in what order they will be executed. The list refers to the names of bundles (which might be parameterized function-like objects).

The order in which you execute bundles can affect the outcome of your promises. In general you should always define variables before you use them.

The `bundlesequence` is like a genetic makeup of a machine. The bundles act like characteristics of the systems. If you want different systems to have different bundlesequences, distinguish them with classes:

```
webservers::

  bundlesequence => { "main", "web" };

others::

  bundlesequence => { "main", "otherstuff" };
```

If you want to add a basic common sequence to all sequences, then use global variable lists to do this:

```
body common control
{
webservers::

  bundlesequence => { @(g.bs), "web" };

others::

  bundlesequence => { @(g.bs), "otherstuff" };

}
```

```
bundle common g
{
vars:

  "bs" slist => { "main", "basic_stuff" };
}
```

Default value:

There is no default value for `bundlesequence`, and the absence of a `bundlesequence` will cause a compilation error. A bundlesequence may also be specified using the `-b` or `--bundlesequence` command line option.

5.1.2 `goal_patterns`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of regular expressions that match promisees/topics considered to be organizational goals

Example:

```
body common control
{
goal_patterns => { "goal_.*", "target.*" };
}
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1.0 (2011)

Used as identifier to mark business and organizational goals in commercial versions of CFEngine. CFEngine uses this to match promisees that represent business goals in promises.

5.1.3 `ignore_missing_bundles`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: If any bundles in the bundlesequence do not exist, ignore and continue

Example:

```
ignore_missing_bundles => "true";
```

Notes:

This authorizes the bundlesequence to contain possibly "nonexistent" pluggable modules. It defaults to false, whereupon undefined bundles cause a fatal error in parsing, and a transition to failsafe mode.

5.1.4 `ignore_missing_inputs`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: If any input files do not exist, ignore and continue

Example:

```
ignore_missing_inputs => "true";
```

Notes:

The inputs lists determines which files are parsed by CFEngine. Normally stringent security checks are made on input files to prevent abuse of the system by unauthorized users. Sometimes however, it is appropriate to consider the automatic plug-in of modules that might or might not exist. This option permits CFEngine to list possible files that might not exist and continue 'best effort' with those that do exist. The default of all Booleans is false, so the normal behaviour is to signal an error if an input is not found.

5.1.5 `inputs`

Type: slist

Allowed input range: `.*`

Synopsis: List of additional filenames to parse for promises

Example:

```
body common control
{
inputs  => {
           "update.cf",
           "library.cf"
           };
}
```

Notes:

The filenames specified are all assumed to be in the same directory as the file which references them (this is usually `$(sys.workdir)/inputs`, but may be overridden by the `-f` or `--file` command line option.

Default value:

There is no default value. If no filenames are specified, no other filenames will be included in the compilation process.

5.1.6 `version`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Scalar version string for this configuration

Example:

```
body common control
{
version => "1.2.3";
}
```

Notes:

The version string is used in error messages and reports.

This string should not contain the colon ':' character, as this has a special meaning in the context of knowledge management. This restriction might be lifted later.

5.1.7 `lastseenexpireafter`

Type: int

Allowed input range: `0,99999999999`
Default value: One week
Synopsis: Number of minutes after which last-seen entries are purged
Example:

```
body common control
{
lastseenexpireafter => "72";
}
```

Notes:

Default time is one week.

5.1.8 output_prefix

Type: string
Allowed input range: (arbitrary string)
Synopsis: The string prefix for standard output
Example:

```
body common control
{
output_prefix => "my_cf3";
}
```

Notes:

On native Windows versions of CFEngine (Nova and above), this string is also prefixed messages in the event log.

5.1.9 domain

Type: string
Allowed input range: `.*`
Synopsis: Specify the domain name for this host
Example:

```
body common control
```

```
{
domain => "example.org";
}
```

Notes:

There is no standard, universal or reliable way of determining the DNS domain name of a host, so it can be set explicitly to simplify discovery and name-lookup.

5.1.10 require_comments

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: Warn about promises that do not have comment documentation
Example:

```
body common control

{
common::

require_comments => "true";
}
```

Notes:

This may be used as a policy Quality Assurance measure, to remind policy makers to properly document their promises. When true, `cf-promises` will report loudly on promises that do not have comments. Variables promises are exempted from this rule, since they may be considered self-documenting.

5.1.11 host_licenses_paid

Type: int
Allowed input range: `0,99999999999`
Default value: 0

Synopsis: The number of licenses that you promise to have paid for by setting this value (legally binding for commercial license)

Example:

```
body common control
{
host_licenses_paid => "1000";
}
```

Notes:

Licensees of the commercial CFEngine releases have to make a promise in acceptance of contract terms by setting this value to the number of licenses they have paid for. This is tallied with the number of licenses granted. This declaration should be placed in all separate configuration files, e.g. 'failsafe.cf', 'promises.cf'.

5.1.12 site_classes

Type: clist

Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`

Synopsis: A list of classes that will represent geographical site locations for hosts. These should be defined elsewhere in the configuration in a classes promise.

Example:

```
body common control
{
site_classes => { "datacenters","datacentres"  }; # locations is by default
}
```

Notes:

History: Was introduced in version 3.2.0, Nova 2.1.0 (2011)

This list is used to match against topics when connecting inferences about host locations in the knowledge map. Normally any CFEngine classes promise whose name is defined as a thing or topic under class `locations::` will be assumed to be a location defining classifier. This list will add alternative class contexts for interpreting location.

5.1.13 syslog_host

Type: string

Allowed input range: `[a-zA-Z0-9_$(){}.:-]+`

Default value: 514

Synopsis: The name or address of a host to which syslog messages should be sent directly by UDP

Example:

```
body common control
{
syslog_host => "syslog.example.org";
syslog_port => "514";
}
```

Notes:

The hostname or IP address of a local syslog service to which all CFEngine's components may promise to send data. This feature is provided in CFEngine Nova and above.

5.1.14 syslog_port

Type: int

Allowed input range: `0,99999999999`

Synopsis: The port number of a UDP syslog service

Example:

```
body common control
{
syslog_host => "syslog.example.org";
syslog_port => "514";
}
```

Notes:

The UDP port of a local syslog service to which all CFEngine's components may promise to send data. This feature is provided in CFEngine Nova and above.

5.1.15 fips_mode

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: Activate full FIPS mode restrictions

Example:

```
body common control
{
fips_mode => "true";
}
```

Notes:

Appears as of Nova 2.0. If CFEngine commercial editions this value may be set to avoid the use of old deprecated algorithms that are no longer FIPS 140-2 compliant. If not set, there is some degree of compatibility with older versions and algorithms. During an upgrade, setting this parameter can cause a lot of recomputation of checksums etc. Government bodies starting with Nova 2.0 or higher should set this to 'true' from the start.

5.2 agent control promises

```
body agent control
{
123_456_789::

  domain => "mydomain.com";

123_456_789_111::

  auditing => "true";

any::

  fullencryption => "true";

}
```

Settings describing the details of the fixed behavioural promises made by `cf-agent`.

5.2.1 abortclasses

Type: slist

Allowed input range: `.*`

Synopsis: A list of classes which if defined lead to termination of cf-agent

Example:

```
 body agent control

  {
  abortclasses => { "danger.*", "should_not_continue" };
  }
```

Notes:

A list of class regular expressions that `cf-agent` will watch out for. If any matching class becomes defined, it will cause the current execution of `cf-agent` to be aborted. This may be used for validation, for example. To handle class expressions, simply create an alias for the expression with a single name.

5.2.2 abortbundleclasses

Type: slist
Allowed input range: `.*`
Synopsis: A list of classes which if defined lead to termination of current bundle
Example:

This example shows how to use the feature to validate input to a method bundle.

```
body common control

{
bundlesequence  => { "testbundle"  };
version => "1.2.3";
}

###########################################

body agent control

{
abortbundleclasses => { "invalid.*" };
}

###########################################

bundle agent testbundle
{
vars:
```

```
 "userlist" slist => { "xyz", "mark", "jeang", "jonhenrik", "thomas", "eben" };

methods:

 "any" usebundle => subtest("$(userlist)");

}

#########################################

bundle agent subtest(user)

{
classes:

  "invalid" not => regcmp("[a-z]{4}","$(user)");

reports:

 !invalid::

  "User name $(user) is valid at exactly 4 letters";

 # abortbundleclasses will prevent this from being evaluated
 invalid::

  "User name $(user) is invalid";
}
```

Notes:

A list of regular expressions for classes, or class expressions that `cf-agent` will watch out for. If any of these classes becomes defined, it will cause the current bundle to be aborted. This may be used for validation, for example.

5.2.3 `addclasses`

Type: slist
Allowed input range: `.*`
Synopsis: A list of classes to be defined always in the current context
Example:

Add classes adds global, literal classes. The only predicates available during the control section are hard-classes.

```
any::

  addclasses => { "My_Organization" }

solaris::

  addclasses => { "some_solaris_alive", "running_on_sunshine" };
```

Notes:

Another place to make global aliases for system hardclasses. Classes here are added unqeuivocally to the system. If classes are used to predicate definition, then they must be defined in terms of global hard classes.

5.2.4 agentaccess

Type: slist

Allowed input range: `.*`

Synopsis: A list of user names allowed to execute cf-agent

Example:

```
 agentaccess => { "mark", "root", "sudo" };
```

Notes:

A list of user names that will be allowed to attempt execution of the current configuration. This is mainly a sanity check rather than a security measure.

5.2.5 agentfacility

Type: (menu option)

Allowed input range:

```
LOG_USER
LOG_DAEMON
LOG_LOCAL0
LOG_LOCAL1
LOG_LOCAL2
LOG_LOCAL3
LOG_LOCAL4
LOG_LOCAL5
```

```
LOG_LOCAL6
LOG_LOCAL7
```

Default value: LOG_USER
Synopsis: The syslog facility for cf-agent
Example:

```
agentfacility => "LOG_USER";
```

Notes:

Sets the agent's syslog facility level. See the manual pages for syslog. This is ignored on Windows, as CFEngine Nova creates event logs.

5.2.6 `allclassesreport`

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Generate allclasses.txt report
Example:

```
body agent control
{
allclassesreport => "true";
}
```

Notes:

History: Was introduced in 3.2.0, Nova 2.1.0 (2011)

This option determines whether state/allclasses.txt file is written to disk during agent execution. This functionality is retained only for CFEngine 2 compatibility as more convenient facilities exist in CFEngine 3 language to achieve similar results.

This option is turned off by default.

5.2.7 `alwaysvalidate`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false flag to determine whether configurations will always be checked before executing, or only after updates

Example:

```
body agent control
{
Min00_05::

  # revalidate once per hour, regardless of change in configuration

  alwaysvalidate => "true";
}
```

Notes:

History: Was introduced in version 3.1.2,Nova 2.0.1 (2010)

The agents `cf-agent`, and `cfserverd` etc can run `cf-promises` to validate inputs before attempting to execute a configuration. As of version 3.1.2 core, this only happens if the configuration file has changed to save CPU cycles. When this attribute is set, `cf-agent` will force a revalidation of the input.

5.2.8 `auditing`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false flag to activate the cf-agent audit log

Example:

```
body agent control
{
auditing  => "true";
}
```

Notes:

If this is set, CFEngine will perform auditing on promises in the current configuration. This means that all details surrounding the verification of the current promise will be recorded in the audit database. The database may be inspected with `cf-report`, or `cfshow` in CFEngine 2.

5.2.9 binarypaddingchar

Type: string
Allowed input range: (arbitrary string)
Default value: space (ASC=32)
Synopsis: Character used to pad unequal replacements in binary editing
Example:

```
body agent control
{
binarypaddingchar => "#";
}
```

Notes:

When editing binary files, it can be dangerous to replace a text string with one that is longer or shorter as byte references and jumps would be destroyed. CFEngine will therefore not allow replacements that are larger in size than the original, but shorter strings can be padded out to the same length.
Default value:

The `binarypaddingchar` defaults to the empty string (i.e., no padding)

5.2.10 bindtointerface

Type: string
Allowed input range: `.*`
Synopsis: Use this interface for outgoing connections
Example:

```
bindtointerface => "192.168.1.1";
```

Notes:

On multi-homed hosts, the server and client can bind to a specific interface for server traffic. The IP address of the interface must be given as the argument, not the device name.

5.2.11 hashupdates

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false whether stored hashes are updated when change is detected in source
Example:

```
body agent control
{
hashupdates => "true";
}
```

Notes:

If 'true' the stored reference value is updated as soon as a warning message has been given. As most changes are benign (package updates etc) this is a common setting.

5.2.12 childlibpath

Type: string
Allowed input range: .*
Synopsis: LD_LIBRARY_PATH for child processes
Example:

```
body agent control
{
childlibpath => "/usr/local/lib:/usr/local/gnu/lib";
```

```
}
```

Notes:

This string may be used to set the internal `LD_LIBRARY_PATH` environment of the agent.

5.2.13 `checksum_alert_time`

Type: int
Allowed input range: `0,60`
Default value: 10 mins
Synopsis: The persistence time for the checksum_alert class
Example:

```
body agent control
{
checksum_alert_time => "30";
}
```

Notes:

When checksum changes trigger an alert, this is registered as a persistent class. This value determines the longevity of that class.

5.2.14 `defaultcopytype`

Type: (menu option)
Allowed input range:

```
               mtime
               atime
               ctime
               digest
               hash
               binary
```

Synopsis: ctime or mtime differ
Example:

```
body agent control
{
#...
defaultcopytype => "digest";
}
```

Notes:

Sets the global default policy for comparing source and image in copy transactions.

5.2.15 dryrun

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: All talk and no action mode
Example:

```
body agent control
{
dryrun => "true";
}
```

Notes:

If set in the configuration, CFEngine makes no changes to a system, only reports what it needs to do.

5.2.16 editbinaryfilesize

Type: int
Allowed input range: `0,99999999999`
Default value: 100000
Synopsis: Integer limit on maximum binary file size to be edited
Example:

```
body agent control
{
edibinaryfilesize => "10M";
```

```
}
```

Notes:

The global setting for the file-editing safety-net for binary files (this value may be overridden on a per-promise basis with `max_file_size`, See Section 7.4.7 [edit_defaults in files], page 253. The default value for `editbinaryfilesize` is `100k`. Note the use of special units is allowed, See Section 2.6 [Datatypes in CFEngine 3], page 24, for a list of permissible suffixes.

When setting limits, the limit on editing binary files should generally be set higher than for text files.

5.2.17 editfilesize

Type: int

Allowed input range: `0,99999999999`

Default value: 100000

Synopsis: Integer limit on maximum text file size to be edited

Example:

```
body agent control
{
editfilesize => "120k";
}
```

Notes:

The global setting for the file-editing safety-net (this value may be overridden on a per-promise basis with `max_file_size`, See Section 7.4.7 [edit_defaults in files], page 253. Note the use of special units is allowed, See Section 2.6 [Datatypes in CFEngine 3], page 24, for a list of permissible suffixes.

5.2.18 environment

Type: slist

Allowed input range: `[A-Za-z0-9_]+=.*`

Synopsis: List of environment variables to be inherited by children

Example:

```
body common control
{
bundlesequence => { "one" };
}
```

```
body agent control
{
environment => { "A=123", "B=456", "PGK_PATH=/tmp"};
}

bundle agent one
{
commands:

  "/usr/bin/env";
}
```

Notes:

This may be used to set the runtime environment of the agent process. The values of environment variables are inherited by child commands. Some interactive programs insist on values being set, e.g.

```
# Required by apt-cache, debian

environment => { "LANG=C"};
```

5.2.19 exclamation

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: true

Synopsis: true/false print exclamation marks during security warnings

Example:

```
body agent control
{
exclamation => "false";
}
```

Notes:

This affects only the output format of warnings.

5.2.20 expireafter

Type: int
Allowed input range: 0,99999999999
Default value: 1 min
Synopsis: Global default for time before on-going promise repairs are interrupted
Example:

```
body action example
{
ifelapsed   => "120";      # 2 hours
expireafter => "240";      # 4 hours
}
```

Notes:

The locking time after which CFEngine will attempt to kill and restart its attempt to keep a promise.

5.2.21 files_single_copy

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of filenames to be watched for multiple-source conflicts
Example:

```
body agent control
{
files_single_copy => { "/etc/.*", "/special/file" };
}
```

Notes:

This list of regular expressions will ensure that files matching the patterns of the list are never copied from more than one source during a single run of `cf-agent`. This may be considered a protection against accidential overlap of copies from diverse remote sources, or as a first-come-first-served disambiguation tool for lazy-evaluation of overlapping file-copy promises.

5.2.22 files_auto_define

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of filenames to define classes if copied

Example:

```
body agent control
{
files_auto_define => { "/etc/syslog\.c.*", "/etc/passwd" };
}
```

Notes:

Classes are automatically defined by the files that are copied. The file is named according to the prefixed 'canonization' of the file name. Canonization means that non-identifier characters are converted into underscores. Thus '`/etc/passwd`' would canonize to '`_etc_passwd`'. The prefix '`auto_`' is added to clarify the origin of the class. Thus in the example the copying of '`/etc/passwd`' would lead to the class '`auto__etc_passwd`' being defined automatically.

5.2.23 `hostnamekeys`

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false label ppkeys by hostname not IP address
Example:

```
body server control
{
hostnamekeys => "true";
}
```

Notes:

Client side choice to base key associations on host names rather than IP address. This is useful for hosts with dynamic addresses.

This feature has been deprecated since 3.1.0. Host identification is now handled transparently.

5.2.24 `ifelapsed`

Type: int

Allowed input range: `0,99999999999`

Default value: 1

Synopsis: Global default for time that must elapse before promise will be rechecked

Example:

```
#local

body action example
{
ifelapsed   => "120";      # 2 hours
expireafter => "240";      # 4 hours
}

# global

body agent control
{
ifelapsed   => "180";      # 3 hours
}
```

Notes:

This overrides the global settings. Promises which take a long time to verify should usually be protected with a long value for this parameter. This serves as a resource 'spam' protection. A CFEngine check could easily run every 5 minutes provided resource intensive operations are not performed on every run. Using time classes like `Hr12` etc., is one part of this strategy; using `ifelapsed` is another which is not tied to a specific time.

5.2.25 `inform`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false set inform level default

Example:

```
body agent control
{
inform => "true";
}
```

Notes:

Equivalent to (and when present, overrides) the command line option '-I'. Sets the default output level 'permanently' within the class context indicated.

Every promiser makes an implicit default promise to use output settings declared using `outputs` promises.

5.2.26 `intermittency`

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: This option is deprecated, does nothing and is kept for backward compatibility
Example:

Notes:

5.2.27 `max_children`

Type: int
Allowed input range: `0,99999999999`
Default value: 1 concurrent agent promise
Synopsis: Maximum number of background tasks that should be allowed concurrently
Example:

```
body runagent control
{
max_children => "10";
}

# or

body agent control
{
max_children => "10";
}
```

Notes:

For the run-agent this represents the maximum number of forked background processes allowed when parallelizing connections to servers. For the agent it represents the number of background jobs allowed concurrently. Background jobs often lead to contention of the disk resources slowing down tasks considerably; there is thus a law of diminishing returns.

5.2.28 maxconnections

Type: int
Allowed input range: `0,99999999999`
Default value: 30 remote queries
Synopsis: Maximum number of outgoing connections to cf-serverd
Example:

```
# client side

body agent control
{
maxconnections => "1000";
}

# server side

body server control
{
maxconnections => "1000";
}
```

Notes:

Watch out for kernel limitations for maximum numbers of open file descriptors which can limit this.

5.2.29 mountfilesystems

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false mount any filesystems promised

Example:

```
body agent control
{
mountfilesystems => "true";
}
```

Notes:

Issues the generic command to mount file systems defined in the file system table.

5.2.30 nonalphanumfiles

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false warn about filenames with no alphanumeric content

Example:

```
body agent control
```

```
{
nonalphanumfiles => "true";
}
```

Notes:

This test is applied in all recursive/depth searches.

5.2.31 `repchar`

Type: string
Allowed input range: .
Default value: _
Synopsis: The character used to canonize pathnames in the file repository
Example:

```
body agent control
{
repchar => "_";
}
```

Notes:

5.2.32 `refresh_processes`

Type: slist
Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`
Synopsis: Reload the process table before verifying the bundles named in this list (lazy evaluation)
Example:

```
body agent control
{
refresh_processes => { "mybundle" };
#refresh_processes => { "none" };
}
```

Notes:

History: Was introduced in version 3.1.3,Nova 2.0.2 (2010)

If this list of regular expressions is non-null and an existing bundle is mentioned or matched in this list, CFEngine will reload the process table at the start of the named bundle, each time is is scheduled. If the list is null, the process list will be reloaded at the start of every scheduled bundle.

In the example above we use a non-empty list with the name 'none'. This is not a reserved word, but as long as there are no bundles with the name 'none' this has the effect of *never* reloading the process table. This keeps improves the efficiency of the agent.

5.2.33 `default_repository`

Type: string

Allowed input range: `"?(/.*)`

Default value: in situ

Synopsis: Path to the default file repository

Example:

```
body agent control
{
default_repository => "/var/cfengine/repository";
}
```

Notes:

If defined the default repository is the location where versions of files altered by CFEngine are stored. This should be understood in relation to the policy for '`backup`' in copying, editing etc. If the backups are time-stamped, this becomes effective a version control repository. See also Section 7.4.17 [repository], page 277 for a way to locally override the global repository.

Note that when a repository is specified, the files are stored using the canonified directory name of the original file, concatenated with the name of the file. So, for example, '`/usr/local/etc/postfix.conf`' would ordinarily be stored in an alternative repository as '`_usr_local_etc_postfix.conf.cfsaved`'.

5.2.34 `secureinput`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false check whether input files are writable by unauthorized users

Example:

```
body agent control
{
secureinput => "true";
}
```

Notes:

If this is set, the agent will not accept an input file that is not owned by a privileged user.

5.2.35 sensiblecount

Type: int
Allowed input range: 0,99999999999
Default value: 2 files
Synopsis: Minimum number of files a mounted filesystem is expected to have
Example:

```
body agent control
{
sensiblecount => "20";
}
```

Notes:

5.2.36 sensiblesize

Type: int
Allowed input range: 0,99999999999
Default value: 1000 bytes
Synopsis: Minimum number of bytes a mounted filesystem is expected to have
Example:

```
body agent control
{
sensiblesize => "20K";
}
```

Notes:

5.2.37 skipidentify

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: Do not send IP/name during server connection because address resolution is broken

Example:

```
body agent control
{
skipidentify => "true";
}
```

Notes:

Hosts that are not registered in DNS cannot supply reasonable credentials for a secondary confirmation of their identity to a CFEngine server. This causes the agent to ignore its missing DNS credentials.

5.2.38 suspiciousnames

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of names to warn about if found during any file search

Example:

```
body agent control
{
suspiciousnames => { ".mo", "lrk3", "rootkit" };
}
```

Notes:

If CFEngine sees these names during recursive (depth) file searches it will warn about them.

5.2.39 syslog

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false switches on output to syslog at the inform level

Example:

```
body agent control
{
syslog => "true";
}
```

Notes:

5.2.40 track_value

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false switches on tracking of promise valuation

Example:

```
body agent control
{
track_value => "true";
}
```

Notes:

If this is true, CFEngine generates a log in 'WORKDIR/state/cf_value.log' of the estmated 'business value' of the system automation as a running log, value_kept, etc. The format of the file is

```
date,sum value kept,sum value repaired,sum value notkept
```

5.2.41 timezone

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of allowed timezones this machine must comply with

Example:

```
body agent control
{
timezone => { "MET", "CET", "GMT+1" };
}
```

Notes:

5.2.42 default_timeout

Type: int

Allowed input range: 0,99999999999

Default value: 10 seconds

Synopsis: Maximum time a network connection should attempt to connect

Example:

```
body agent control
{
```

```
default_timeout => "10";
}
```

Notes:

The time is in seconds. It is not a guaranteed number, since it depends on system behaviour. under Linux, the kernel version plays a role, since not all system calls seem to respect the signals.

5.2.43 `verbose`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false switches on verbose standard output

Example:

```
body agent control
{
verbose => "true";
}
```

Notes:

Equivalent to (and when present, overrides) the command line option '`-v`'. Sets the default output level 'permanently' for this promise.

Every promiser makes an implicit default promise to use output settings declared using `outputs` promises.

5.3 `server` control promises

```
body server control

{
allowconnects         => { "127.0.0.1" , "::1" ,  ".*\.example\.org" };
allowallconnects      => { "127.0.0.1" , "::1" ,  ".*\.example\.org" };

# Uncomment me under controlled circumstances
#trustkeysfrom         => { "127.0.0.1" , "::1" ,  ".*\.example\.org" };
}
```

Settings describing the details of the fixed behavioural promises made by `cf-serverd`. Server controls are mainly about determining access policy for the connection protocol: i.e. access to the server itself. Access to specific files must be granted in addition.

5.3.1 `allowallconnects`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames that may have more than one connection to the server port

Example:

```
allowallconnects      => {
                         "127.0.0.1",
                         "::1",
                         "200\.1\.10\..*",
                         "host\.domain\.tld",
                         "host[0-9]+\.domain\.com"
                         };
```

Notes:

This list of regular expressions matches hosts that are allowed to connect an umlimited number of times up to the maximum connection limit. Without this, a host may only connect once (which is a very strong constraint, as the host must wait for the TCP FIN_WAIT to expire before reconnection can be attempted).

In CFEngine 2 this corresponds to `AllowMultipleConnectionsFrom`.

Note that `127.0.0.1` is a regular expression (i.e., “127 any character 0 any character 0 any character 1”), but this will only match the IP address `127.0.0.1`. Take care with IP addresses and domain names, as the hostname regular expression `www.domain.com` will potentially match more than one hostname (e.g., `wwwxdomain.com`, in addition to the desired hostname `www.domain.com`).

5.3.2 `allowconnects`

Type: slist

Allowed input range: (arbitrary string)
Synopsis: List of IPs or hostnames that may connect to the server port
Example:

```
allowconnects         => {
                         "127.0.0.1",
                         "::1",
                         "200\.1\.10\..*",
                         "host\.domain\.tld",
                         "host[0-9]+\.domain\.com"
                         };
```

Notes:

If a client's identity matches an entry in this list it is granted to permission to send data to the server port. Clients who are not in this list may not connect or send data to the server.

See also the warning about regular expressions in `allowallconnects`.

5.3.3 allowusers

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of usernames who may execute requests from this server
Example:

```
allowusers => { "cfengine", "root" };
```

Notes:

The usernames listed in this list are those asserted as public key identities during client-server connections. These may or may not correspond to system identities on the server-side system.

5.3.4 auditing

Type: (menu option)
Allowed input range:

```
true
false
yes
no
```

```
on
off
```

Default value: false
Synopsis: true/false activate auditing of server connections
Example:

```
body agent control
{
auditing  => "true";
}
```

Notes:

If this is set, CFEngine will perform auditing on promises in the current configuration. This means that all details surrounding the verification of the current promise will be recorded in the audit database. The database may be inspected with `cf-report`, or `cfshow` in CFEngine 2.

5.3.5 `bindtointerface`

Type: string
Allowed input range: (arbitrary string)
Synopsis: IP of the interface to which the server should bind on multi-homed hosts
Example:

```
bindtointerface => "192.168.1.1";
```

Notes:

On multi-homed hosts, the server and client can bind to a specific interface for server traffic. The IP address of the interface must be given as the argument, not the device name.

5.3.6 `cfruncommand`

Type: string
Allowed input range: `.+`
Synopsis: Path to the cf-agent command or cf-execd wrapper for remote execution
Example:

```
body server control
```

```
{
cfruncommand => "/var/cfengine/bin/cf-agent";
}
```

Notes:

It is normal for this to point to the location of `cf-agent` but it could also point to the `cf-execd`, or even another program or shell command at your own risk.

5.3.7 call_collect_interval

Type: int

Allowed input range: `0,99999999999`

Synopsis: The interval in minutes in between collect calls to the policy hub offering a tunnel for report collection (Enterprise)

Example:

```
call_collect_interval => "5";
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

If option time is set, it causes the server daemon to peer with a policy hub by attempting a connection at regular intervals of the value of the parameter in minutes.

This feature is designed to allow Enterprise report collection from hosts that are not directly addressable from a hub data-aggregation process. For example, if some of the clients of a policy hub are behind a network address translator then the hub is not able to open a channel to address them directly. The effect is to place a 'collect call' with the policy hub.

If this option is set, the client's cf-serverd will 'peer' with the server daemon on a policy hub. This means that, cf-serverd on an unreachable (e.g. NATed) host will attempt to report in to the cf-serverd on its assigned policy hub and offer it a short time window in which to download reports over the established connection. The effect is to establish a temporary secure tunnel between hosts, initiated from the satellite host end. The connection is made in such a way that host autonomy is not compromised. Either hub may refuse or decline to play their role at any time, in the usual way (avoiding DOS attacks). Normal access controls must be set for communication in both directions.

Collect calling cannot be as efficient as data collection by the cf-hub, as the hub is not able to load balance. Hosts that use this approach should exclude themselves from the cf-hub data collection.

The sequence of events is this:

- Satellite cf-serverd connects to its registered policy hub

- The satellite indentifies itself to authentication and access control and sends a collect-call 'pull' request to the hub
- The hub might honour this, if the access control grants access.
- If access is granted, the hub has `collect_window` seconds to initiate a query to the satellite for its reports.
- The policy hub indentifies itself to authentication and access control and sends a query request to the hub to collect the reports.
- When finished the satellite closes the tunnel.

The full configuration would look something like this

```
#########################################################
# Server config
#########################################################

body server control

{
allowconnects         => { "10.10.10" , "::1" };
allowallconnects      => { "10.10.10" , "::1" };
trustkeysfrom         => { "10.10.10" , "::1" };

call_collect_interval => "5";
}

#########################################################

bundle server access_rules()

{
access:

  policy_hub::

   "collect_calls"
       resource_type => "query",
             admit   => { "10.10.10" }; # the apparent NAT address of the satellite

  satellite_hosts::

    "delta"
            comment => "Grant access to cfengine hub to collect report deltas",
      resource_type => "query",
            admit   => { "policy_hub" };

    "full"
            comment => "Grant access to cfengine hub to collect full report dump",
```

```
      resource_type => "query",
            admit   => { "policy_hub"  };
}
```

5.3.8 collect_window

Type: int

Allowed input range: `0,99999999999`

Synopsis: A time in seconds that a collect-call tunnel remains open to a hub to attempt a report transfer before it is closed (Enterprise)

Example:

```
collect_window => "15";
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

The time is measured in seconds, default value 10s.

5.3.9 denybadclocks

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Default value: true

Synopsis: true/false accept connections from hosts with clocks that are out of sync

Example:

```
body server control
{
#..
denybadclocks => "true";
}
```

Notes:

A possible form of attack on the fileserver is to request files based on time by setting the clocks incorrectly. This option prevents connections from clients whose clocks are drifting too far from the server clock (where "too far" is currently defined as "more than an hour off"). This serves as a warning about clock asynchronization and also a protection against Denial of Service attempts based on clock corruption.

5.3.10 denyconnects

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of IPs or hostnames that may NOT connect to the server port
Example:

```
body server control
{
denyconnects => { "badhost\.domain\.evil", "host3\.domain\.com" };
}
```

Notes:

Hosts or IP addresses that are explicitly denied access. This should only be used in special circumstances. One should never grant generic access to everything and then deny special cases. Since the default server behaviour is to grant no access to anything, this list is unnecessary unless you have already granted access to some set of hosts using a generic pattern, to which you intend to make an exception.

See also the warning about regular expressions in `allowallconnects`.

5.3.11 dynamicaddresses

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of IPs or hostnames for which the IP/name binding is expected to change
Example:

```
body server control
{
dynamicaddresses => { "dhcp_.*" };
}
```

Notes:

The addresses or hostnames here are expected to have non-permanent address-name bindings, we must therefor work harder to determine whether hosts credentials are trusted by looking for existing public keys in files that do not match the current hostname or IP.

This feature has been deprecated since 3.1.0. This is now handled transparently.

5.3.12 hostnamekeys

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false store keys using hostname lookup instead of IP addresses

Example:

```
body server control
{
hostnamekeys => "true";
}
```

Notes:

Client side choice to base key associations on host names rather than IP address. This is useful for hosts with dynamic addresses.

This feature has been deprecated since 3.1.0. Host identification is now handled transparently.

5.3.13 keycacheTTL

Type: int

Allowed input range: `0,99999999999`

Default value: 24

Synopsis: Maximum number of hours to hold public keys in the cache

Example:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

```
body server control
{
```

```
keycacheTTL => "24";
}
```

Notes:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

5.3.14 logallconnections

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false causes the server to log all new connections to syslog
Example:

```
body server control
{
logallconnections => "true";
}
```

Notes:

If set, the server will record connection attempts in syslog.

5.3.15 logencryptedtransfers

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false log all successful transfers required to be encrypted
Example:

```
body server control
{
logencryptedtransfers => "true";
}
```

Notes:

If true the server will log all transfers of files which the server requires to encrypted in order to grant access (see `ifencrypted`) to syslog. These files are deemed to be particularly sensitive.

5.3.16 `maxconnections`

Type: int
Allowed input range: `0,99999999999`
Default value: 30 remote queries
Synopsis: Maximum number of connections that will be accepted by cf-serverd
Example:

```
# client side

body agent control
{
maxconnections => "1000";
}

# server side

body server control
{
maxconnections => "1000";
}
```

Notes:

Watch out for kernel limitations for maximum numbers of open file descriptors which can limit this.

5.3.17 `port`

Type: int

Allowed input range: `1024,99999`

Default value: 5308

Synopsis: Default port for cfengine server

Example:

```
body hub control
{
port => "5308";
}

body server control
{
specialhost::
 port => "5308";

!specialhost::
 port => "5308";
}
```

Notes:

The standard or registered port number is tcp/5308. CFEngine does not presently use its registered udp port with the same number, but this could change in the future.

Changing the standard port number is not recommended practice. You should not do it without a good reason.

5.3.18 `serverfacility`

Type: (menu option)

Allowed input range:

```
LOG_USER
LOG_DAEMON
LOG_LOCAL0
LOG_LOCAL1
LOG_LOCAL2
LOG_LOCAL3
LOG_LOCAL4
LOG_LOCAL5
LOG_LOCAL6
LOG_LOCAL7
```

Default value: LOG_USER

Synopsis: Menu option for syslog facility level

Example:

```
body server control
{
serverfacility => "LOG_USER";
}
```

Notes:

See syslog notes.

5.3.19 skipverify

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames for which we expect no DNS binding and cannot verify

Example:

```
body server control
{
skipverify => { "special_host.*", "192.168\..*" };
}
```

Notes:

Server side decision to ignore requirements of DNS identity confirmation.

See also the warning about regular expressions in `allowallconnects`.

5.3.20 trustkeysfrom

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs from whom we accept public keys on trust

Example:

```
body server control
{
trustkeysfrom => { "10\.0\.1\.1", "192\.168\..*"};
}
```

Notes:

If connecting clients' public keys have not already been trusted, this allows us to say 'yes' to accepting the keys on trust. Normally this should be an empty list except in controlled circumstances.

See also the warning about regular expressions in `allowallconnects`.

5.3.21 `listen`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: true

Synopsis: true/false enable server deamon to listen on defined port

Example:

```
body server control
{

  listening_host_context::
    listen => "true";

  !listening_host_context::
    listen => "false";
}
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0 (2012)

This attribute allows to disable `cf-serverd` from listening on any port. Should be used in conjunction with `call_collect_interval`.

This setting only applies to CFEngine clients, the policy hub will not be affected. Changing this setting requires a restart of `cf-serverd` for the change to take effect.

5.4 `monitor` control promises

```
body monitor control()
   {
   #version => "1.2.3.4";

   forgetrate => "0.7";
   tcpdump => "false";
   tcpdumpcommand => "/usr/sbin/tcpdump -i eth1 -n -t -v";

   }
```

Settings describing the details of the fixed behavioural promises made by `cf-monitord`. The system defaults will be sufficient for most users. This configurability potential, however, will be a key to developing the integrated monitoring capabilities of CFEngine.

5.4.1 `forgetrate`

Type: real
Allowed input range: `0,1`
Default value: 0.6
Synopsis: Decimal fraction [0,1] weighting of new values over old in 2d-average computation
Example:

```
body monitor control
{
forgetrate => "0.7";
}
```

Notes:

Configurable settings for the machine-learning algorithm that tracks system behaviour. This is only for expert users. This parameter effectively determines (together with the monitoring rate) how quickly CFEngine forgets its previous history.

5.4.2 `monitorfacility`

Type: (menu option)
Allowed input range:

```
LOG_USER
LOG_DAEMON
LOG_LOCAL0
LOG_LOCAL1
LOG_LOCAL2
LOG_LOCAL3
```

```
LOG_LOCAL4
LOG_LOCAL5
LOG_LOCAL6
LOG_LOCAL7
```

Default value: LOG_USER

Synopsis: Menu option for syslog facility

Example:

```
body monitor control
{
monitorfacility => "LOG_USER";
}
```

Notes:

See notes for syslog.

5.4.3 histograms

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: true

Synopsis: Ignored, kept for backward compatibility

Example:

```
body monitor control
{
histograms => "true";
}
```

Notes:

`cf-monitord` now always keeps histograms information, so this option is a no-op kept for backward compatibility. It used to cause CFEngine to learn the conformally transformed distributions of fluctuations about the mean.

5.4.4 `tcpdump`

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false use tcpdump if found
Example:

```
body monitor control
{
tcpdump => "true";
}
```

Notes:

Interface with TCP stream if possible.

5.4.5 `tcpdumpcommand`

Type: string
Allowed input range: `"?(/.*)`
Synopsis: Path to the tcpdump command on this system
Example:

```
body monitor control
{
tcpdumpcommand => "/usr/sbin/tcpdump -i eth1";
}
```

Notes:

If this is defined, the monitor will try to interface with the TCP stream and monitor generic package categories for anomalies.

5.5 runagent control promises

```
body runagent control
{
# default port is 5308

hosts => { "127.0.0.1:5308", "eternity.iu.hio.no:80", "slogans.iu.hio.no" };

#output_to_file => "true";
}
```

Settings describing the details of the fixed behavioural promises made by `cf-runagent`. The most important parameter here is the list of hosts that the agent will poll for connections. This is easily read in from a file list, however when doing so always have a stable input source that does not depend on the network (including a database or directory service) in any way: introducing such dependencies makes configuration brittle.

5.5.1 hosts

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host or IP addresses to attempt connection with

Example:

```
body runagent control
{
network1::
  hosts => { "host1.example.org", "host2", "host3" };

network2::
  hosts => { "host1.example.com", "host2", "host3" };
}
```

Notes:

The complete list of contactable hosts. The values may be either numerical IP addresses or DNS names, optionally suffixed by a ':' and a port number. If no port number is given, the default CFEngine port 5308 is assumed.

5.5.2 port

Type: int

Allowed input range: `1024,99999`

Default value: 5308

Synopsis: Default port for cfengine server

Example:

```
body hub control
{
port => "5308";
}

body server control
{
specialhost::
 port => "5308";

!specialhost::
 port => "5308";
}
```

Notes:

The standard or registered port number is tcp/5308. CFEngine does not presently use its registered udp port with the same number, but this could change in the future.

Changing the standard port number is not recommended practice. You should not do it without a good reason.

5.5.3 force_ipv4

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false force use of ipv4 in connection
Example:

```
body copy_from example
{
force_ipv4 => "true";
}
```

Notes:

IPv6 should be harmless to most users unless you have a partially or misconfigured setup.

5.5.4 trustkey

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false automatically accept all keys on trust from servers
Example:

```
body copy_from example
{
trustkey => "true";
}
```

Notes:

If the server's public key has not already been trusted, this allows us to accept the key in automated key-exchange.

Note that, as a simple security precaution, trustkey should normally be set to '`false`', to avoid key exchange with a server one is not one hundred percent sure about, though the risks for a client are rather low. On the server-side however, trust is often granted to many clients or to a whole network in

which possibly unauthorized parties might be able to obtain an IP address, thus the trust issue is most important on the server side.

As soon as a public key has been exchanged, the trust option has no effect. A machine that has been trusted remains trusted until its key is manually revoked by a system administrator. Keys are stored in 'WORKDIR/ppkeys'.

5.5.5 encrypt

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false encrypt connections with servers

Example:

```
body copy_from example
{
servers  => { "remote-host.example.org" };
encrypt => "true";
}
```

Notes:

Client connections are encrypted with using a Blowfish randomly generated session key. The intial connection is encrypted using the public/private keys for the client and server hosts.

5.5.6 background_children

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false parallelize connections to servers
Example:

```
body runagent control
{
background_children => "true";
}
```

Notes:

Causes the runagent to attempt parallelized connections to the servers.

5.5.7 max_children

Type: int
Allowed input range: 0,99999999999
Default value: 50 runagents
Synopsis: Maximum number of simultaneous connections to attempt
Example:

```
body runagent control
{
max_children => "10";
}

# or

body agent control
{
max_children => "10";
}
```

Notes:

For the run-agent this represents the maximum number of forked background processes allowed when parallelizing connections to servers. For the agent it represents the number of background jobs allowed concurrently. Background jobs often lead to contention of the disk resources slowing down tasks considerably; there is thus a law of diminishing returns.

5.5.8 output_to_file

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false whether to send collected output to file(s)
Example:

```
body runagent control
{
output_to_file => "true";
}
```

Notes:

Filenames are chosen automatically and placed in the '`WORKDIR/outputs/`*`hostname`*`_runagent.out`'.

5.5.9 `output_directory`

Type: string
Allowed input range: `"?(/.*)`
Synopsis: Directory where the output is stored
Example:

```
body runagent control
{
output_directory => "/tmp/run_output";
}
```

Notes:

History: Was introduced in version 3.2.0, Nova 2.1.0 (2011)
Defines the location for parallelized output to be saved when running `cf-runagent` in parallel mode.

5.5.10 `timeout`

Type: int
Allowed input range: `1,9999`

Synopsis: Connection timeout, sec
Example:

```
body runagent control
{
timeout => "10";
}
```

Notes:

Timeout in seconds.

5.6 executor control promises

```
body executor control

{
splaytime  => "5";
mailto     => "cfengine@example.org";
mailfrom   => "cfengine@$(host).example.org";
smtpserver => "localhost";
schedule   => { "Min00_05", "Min30_35" }
}
```

These body settings determine the behaviour of `cf-execd`, including scheduling times and output capture to 'WORKDIR/outputs' and relay via email. Note that the `splaytime` and `schedule` parameters are now coded here rather than (as previously) in the agent.

5.6.1 splaytime

Type: int
Allowed input range: 0,99999999999
Default value: 0
Synopsis: Time in minutes to splay this host based on its name hash
Example:

```
body executor control
{
```

```
splaytime => "2";
}
```

Notes:

Whenever any class listed in the `schedule` attribute is present, `cf-execd` can schedule an execution of `cf-agent`. The actual execution will be delayed an integer number of seconds between 0-`splaytime` minutes. The specific amount of delay for "this" host is based on a hash of the hostname. Thus a collection of hosts will all execute at different times, and surges in network traffic can be avoided.

A rough rule of thumb for scaling of small updates is set the splay time between 1-5 minutes for up a few thousand hosts. The splaytime should not be set to a value larger than the `cf-execd` scheduling interval, else multiple clients might contend for data.

Default value:

The default value is 0 minutes.

See also: The `splayclass()` function for a task-specific means for setting splay times.

5.6.2 `mailfrom`

Type: string

Allowed input range: `.*@.*`

Synopsis: Email-address cfengine mail appears to come from

Example:

```
body executor control
{
mailfrom => "mrcfengine@example.org";
}
```

Notes:

5.6.3 `mailto`

Type: string

Allowed input range: `.*@.*`

Synopsis: Email-address cfengine mail is sent to

Example:

```
body executor control
{
```

```
mailto => "cfengine_alias@example.org";
}
```

Notes:

The address to whom email is sent if an smtp host is configured.

5.6.4 smtpserver

Type: string

Allowed input range: `.*`

Synopsis: Name or IP of a willing smtp server for sending email

Example:

```
body executor control
{
smtpserver => "smtp.example.org";
}
```

Notes:

This should point to a standard port 25 server without encyption. If you are running secured or encrypted email then you should run a mail relay on localhost and point this to localhost.

5.6.5 mailmaxlines

Type: int

Allowed input range: `0,1000`

Default value: 30

Synopsis: Maximum number of lines of output to send by email

Example:

```
body executor control
{
mailmaxlines => "100";
}
```

Notes:

This limit prevents anomalously large outputs from clogging up a system administrator's mailbox. The output is truncated in the email report, but the complete original transcript is stored in 'WORKDIR/outputs/*' where it can be viewed on demand. A reference to the appropriate file is given.

5.6.6 schedule

Type: slist

Allowed input range: (arbitrary string)

Synopsis: The class schedule used by cf-execd for activating cf-agent

Example:

```
body executor control
{
schedule => { "Min00", "(Evening|Night).Min15_20", "Min30", "(Evening|Night).Min45_50" };
}
```

Notes:

The list should contain class expressions comprised of classes which are visible to the cf-execd daemon. In principle, any defined class expression will cause the daemon to wake up and schedule the execution of the cf-agent. In practice, the classes listed in the list are usually date- and time-based.

The actual execution of cf-agent may be delayed by splaytime, and may be deferred by promise caching and the value of ifelapsed. Note also that the effectiveness of the splayclass function may be affected by changing the schedule.

Default value:

```
schedule => { "Min00", "Min05", "Min10", "Min15", "Min20", "Min25",
               "Min30", "Min35", "Min40", "Min45", "Min50", "Min55" };
```

5.6.7 executorfacility

Type: (menu option)

Allowed input range:

```
LOG_USER
LOG_DAEMON
LOG_LOCAL0
LOG_LOCAL1
LOG_LOCAL2
LOG_LOCAL3
LOG_LOCAL4
LOG_LOCAL5
LOG_LOCAL6
```

LOG_LOCAL7

Default value: LOG_USER

Synopsis: Menu option for syslog facility level

Example:

```
body executor control
{
executorfacility => "LOG_USER";
}
```

Notes:

See the syslog manual pages.

5.6.8 exec_command

Type: string

Allowed input range: "?(/.*)

Synopsis: The full path and command to the executable run by default (overriding builtin)

Example:

```
exec_command => "$(sys.workdir)/bin/cf-agent -f failsafe.cf && $(sys.workdir)/bin/cf-agent";
```

Notes:

The command is run in a shell encapsulation so pipes and shell symbols may be used if desired. Unlike, CFEngine 2, CFEngine 3 does not automatically run a separate update sequence before its normal run. This can be handled using the approach in the example above.

5.7 knowledge control promises

```
body knowledge control

{
query_output    => "html";
}
```

Settings describing the details of the fixed behavioural promises made by `cf-know`. These parameters control the way in which knowledge data are stored and retrieved from a relational database and the output format of the queries.

5.7.1 build_directory

Type: string

Allowed input range: `.*`

Default value: Current working directory

Synopsis: The directory in which to generate output files

Example:

```
body knowledge control

{
#..
build_directory => "/tmp/builddir";
}

body reporter control

{
#..
build_directory => "/tmp/builddir";
}
```

Notes:

The directory where all auto-generated textual output is placed by `cf-report`. This includes manual generation, ontology and topic map data.

5.7.2 document_root

Type: string

Allowed input range: `.*`

Synopsis: The directory in which the web root resides

Example:

```
body knowledge control
```

```
{
document_root => "/srv/www/htdocs";
}
```

Notes:

The local file path of the system's web document root.

5.7.3 generate_manual

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false
Synopsis: true/false generate texinfo manual page skeleton for this version
Example:

```
body knowledge control
{
generate_manual => "true";
}
```

Notes:

Auto-creates a manual based on the self-documented code. As the promise syntax is extended the manual self-heals. The resulting manual is generated in Texinfo format, from which all other formats can be generated.

5.7.4 graph_directory

Type: string
Allowed input range: `"?(/.*)`
Synopsis: Path to directory where rendered .png files will be created
Example:

```
body knowledge control
{
graph_directory => "/tmp/output";
}
```

Notes:

A separate location where the potentially large number of '.png' visualizations of a knowledge representation are pre-compiled. This feature only works if the necessary graphics libraries are present.

5.7.5 graph_output

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false generate png visualization of topic map if possible (requires lib)

Example:

```
body knowledge control

{
# fix/override -g option
graph_output => "true";
}
```

Notes:

Equivalent to the use of the '-g' option for `cf-know`.

5.7.6 html_banner

Type: string

Allowed input range: (arbitrary string)

Synopsis: HTML code for a banner to be added to rendered in html after the header

Example:

```
body knowledge control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}

body reporter control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}
```

Notes:

This content is cited when generating HTML output from the knowledge agent.

5.7.7 `html_footer`

Type: string
Allowed input range: (arbitrary string)
Synopsis: HTML code for a page footer to be added to rendered in html before the end body tag
Example:

```
body reporter control
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}

body knowledge control
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}
```

Notes:

This allows us to cite HTML code for formatting reports generated by the reporting and knowledge agents.

5.7.8 `id_prefix`

Type: string
Allowed input range: `.*`

Synopsis: The LTM identifier prefix used to label topic maps (used for disambiguation in merging)
Example:

```
body knowledge control
{
id_prefix => "unique_prefix";
}
```

Notes:

Use to disambiguate indentifiers for a successful merging of topic maps, especially in Linear Topic Map (LTM) format using third party tools such as Ontopia's Omnigator.

5.7.9 manual_source_directory

Type: string
Allowed input range: `"?(/.*)`
Synopsis: Path to directory where raw text about manual topics is found (defaults to build_directory)
Example:

```
body knowledge control
{
manual_source => "/path/cfengine_manual_commentary";
}
```

Notes:

This is used in the self-healing documentation. The directory points to a location where the Texinfo sources for per-section commentary is maintained.

5.7.10 query_engine

Type: string
Allowed input range: (arbitrary string)
Synopsis: Name of a dynamic web-page used to accept and drive queries in a browser
Example:

```
body knowledge control
{
```

```
query_engine => "http://www.example.org/script.php";
}

body reporter control
{
query_engine => "http://www.example.org/script.pl";
}
```

Notes:

When displaying topic maps in HTML format, `cf-know` will render each topic as a link to this URL with the new topic as an argument. Thus it is possible to make a dynamic web query by embedding CFEngine in the web page as system call and passing the argument to it.

5.7.11 query_output

Type: (menu option)

Allowed input range:

```
html
text
```

Synopsis: Menu option for generated output format

Example:

```
body knowledge control
{
query_output => "html";
}
```

Notes:

5.7.12 sql_type

Type: (menu option)

Allowed input range:

```
mysql
postgres
```

Synopsis: Menu option for supported database type

Example:

```
body knowledge control
{
sql_type => "mysql";
}
```

Notes:

5.7.13 sql_database

Type: string
Allowed input range: (arbitrary string)
Synopsis: Name of database used for the topic map
Example:

```
body knowledge control
{
sql_database => "cfengine_knowledge_db";
}
```

Notes:

The name of an SQL database for caching knowledge.

5.7.14 sql_owner

Type: string
Allowed input range: (arbitrary string)
Synopsis: User id of sql database user
Example:

```
body knowledge control
{
sql_owner => "db_owner";
}
```

Notes:

Part of the credentials for opening the database. This depends on the type of database.

5.7.15 sql_passwd

Type: string

Allowed input range: (arbitrary string)

Synopsis: Embedded password for accessing sql database

Example:

```
body knowledge control
{
sql_passwd => "";
}
```

Notes:

Part of the credentials for connecting to the database server. This is system dependent. If the server host is localhost a password might not be required.

5.7.16 sql_server

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name or IP of database server (or localhost)

Example:

```
body knowledge control
{
sql_server => "localhost";
}
```

Notes:

The host name of IP address of the server. The default is to look on the localhost.

5.7.17 sql_connection_db

Type: string

Allowed input range: (arbitrary string)

Synopsis: The name of an existing database to connect to in order to create/manage other databases

Example:

```
body knowledge control
{
sql_connection_db => "mysql";
}
```

Notes:

In order to create a database on a database server (all of which practice voluntary cooperation), one has to be able to connect to the server, however, without an existing database this is not allowed. Thus, database servers provide a default database that can be connected to in order to thereafter create new databases. These are called `postgres` and `mysql` for their respective database servers.

For the knowledge agent, this setting is made in the control body, for database verification promises, it is made in the `database_server` body.

5.7.18 style_sheet

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a style-sheet to be used in rendering html output (added to headers)

Example:

```
body knowledge control
{
style_sheet => "http://www.example.org/css/sheet.css";
}

body reporter control
{
style_sheet => "http://www.example.org/css/sheet.css";
}
```

Notes:

For formatting the HTML generated output of `cf-know`.

5.7.19 view_projections

Type: (menu option)

Allowed input range:

```
true
false
```

```
yes
no
on
off
```

Default value: false

Synopsis: Perform view-projection analytics in graph generation

Example:

```
body knowledge control
{
view_projections => "true";
}
```

Notes:

If this is set to true, CFEngine Nova computes additional graphical representations in its knowledge map, representing causal dependencies between CFEngine promises.

5.8 `reporter` control promises

```
body reporter control
{
reports => { "performance", "last_seen", "monitor_history" };
build_directory => "/tmp/nerves";
report_output => "html";
}
```

Determines a list of reports to write into the build directory. The format may be in text, html or xml format. The reporter agent `cf-report` replaces both `cfshow` and `cfenvgraph`. It no longer produces output to the console.

Some reports are only available in enterprise level versions of CFEngine.

5.8.1 `aggregation_point`

Type: string

Allowed input range: `"?(/.*)`

Synopsis: The root directory of the data cache for CMDB aggregation

Example:

```
body reporter control
{
aggregation_point => "/srv/www/htdocs/reports";
}
```

Notes:

This feature is only used in enterprise level versions of CFEngine. It specifies the directory where reports from multiple hosts are to be aggregated in sub-directories. This should be somewhere under the document root of the web server for the CFEngine knowledge base in order to make the reports browsable.

5.8.2 auto_scaling

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: true

Synopsis: true/false whether to auto-scale graph output to optimize use of space

Example:

```
body reporter control
{
auto_scaling => "true";
}
```

Notes:

Automatic scaling is the default.

5.8.3 build_directory

Type: string

Allowed input range: `.*`

Default value: Current working directory

Synopsis: The directory in which to generate output files

Example:

```
body knowledge control

{
#..
build_directory => "/tmp/builddir";
}

body reporter control

{
#..
build_directory => "/tmp/builddir";
}
```

Notes:

The directory where all auto-generated textual output is placed by `cf-report`. This includes manual generation, ontology and topic map data.

5.8.4 csv2xml

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of csv formatted files in the build directory to convert to simple xml

Example:

```
body reporter control
{
csv2xml => { "myreport.csv", "custom_report.csv"  };
}
```

Notes:

CSV files are easy to generate in CFEngine from individual promise logging functions. XML is not easily generated due to its hierarchical structure. This function allows `cf-report` to convert a CSV file into pidgin XML for convenience. The schema has the general form:

```
<output>
 <line> <one>...</one> <two>...</two> ... </line>
```

```
 <line> <one>...</one> <two>...</two> ... </line>
</output>
```

5.8.5 error_bars

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Default value: true

Synopsis: true/false whether to generate error bars on graph output

Example:

```
body reporter control
{
error_bars => "true";
}
```

Notes:

The default is to produce error bars. Without error bars from CFEngine's machine learning data there is no way to assess the significance of an observation about the system, i.e. whether it is normal or anomalous.

5.8.6 html_banner

Type: string

Allowed input range: (arbitrary string)

Synopsis: HTML code for a banner to be added to rendered in html after the header

Example:

```
body knowledge control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}
```

```
body reporter control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}
```

Notes:

This content is cited when generating HTML output from the knowledge agent.

5.8.7 `html_embed`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: If true, no header and footer tags will be added to html output

Example:

```
body reporter control
{
html_embed => "true";
}
```

Notes:

Embedded HTML means something that could be put into a frame or table, without html or body tags, headers footers etc.

5.8.8 `html_footer`

Type: string

Allowed input range: (arbitrary string)

Synopsis: HTML code for a page footer to be added to rendered in html before the end body tag

Example:

```
body reporter control
```

```
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}

body knowledge control
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}
```

Notes:

This allows us to cite HTML code for formatting reports generated by the reporting and knowledge agents.

5.8.9 query_engine

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a dynamic web-page used to accept and drive queries in a browser

Example:

```
body knowledge control
{
query_engine => "http://www.example.org/script.php";
}

body reporter control
{
query_engine => "http://www.example.org/script.pl";
}
```

Notes:

When displaying topic maps in HTML format, `cf-know` will render each topic as a link to this URL with the new topic as an argument. Thus it is possible to make a dynamic web query by embedding CFEngine in the web page as system call and passing the argument to it.

5.8.10 reports

Type: (option list)

Allowed input range:

```
all
audit
performance
all_locks
active_locks
hashes
classes
last_seen
monitor_now
monitor_history
monitor_summary
compliance
setuid
file_changes
installed_software
software_patches
value
variables
```

Default value: none

Synopsis: A list of reports that may be generated

Example:

```
body reporter control
{
reports => { "performance", "classes"  };
}
```

Notes:

A list of report types that can be generated by this agent. The listed items from `compliance` onward are available only Enterprise editions of CFEngine.

The keyword '`all`' can be used to get all reports except the audit and locking reports. The latter are large and unwieldy and need specific confirmation.

5.8.11 `report_output`

Type: (menu option)

Allowed input range:

```
csv
html
text
```

xml

Default value: none

Synopsis: Menu option for generated output format. Applies only to text reports, graph data remain in xydy format.

Example:

```
body reporter control
{
report_output => "html";
}
```

Notes:

Sets the output format of embedded database reports.

5.8.12 style_sheet

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a style-sheet to be used in rendering html output (added to headers)

Example:

```
body knowledge control
{
style_sheet => "http://www.example.org/css/sheet.css";
}

body reporter control
{
style_sheet => "http://www.example.org/css/sheet.css";
}
```

Notes:

For formatting the HTML generated output of `cf-know`.

5.8.13 time_stamps

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false whether to generate timestamps in the output directory name

Example:

```
body reporter control
{
time_stamps => "true";
}
```

Notes:

This option is only necessary with the default build directory. This can be used to keep snapshots of the system but it will result in a lot of storage be consumed. For most purposes CFEngine is programmed to forget the past at a predictable rate and there is no need to override this.

5.9 hub control promises

```
body hub control
{
export_zenoss => "/var/www/reports/summary.z";
}
```

5.9.1 export_zenoss

Type: string

Allowed input range: `.+`

Synopsis: Generate report for Zenoss integration

Example:

```
body hub control
```

```
{
am_policy_hub::

  export_zenoss => "/var/www/reports/summary.z";
}
```

Notes:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

For integration with the Zenoss monitoring software.

5.9.2 exclude_hosts

Type: slist
Allowed input range: (arbitrary string)
Synopsis: A list of IP addresses of hosts to exclude from report collection
Example:

```
body hub control
{
exclude_hosts => { "192.168.12.21", "10.10", "10.12.*" };
}
```

Notes:

History: Was introduced in 3.3.0, Nova 2.1.1 (2011)

In commercial CFEngine editions, this list of IP addresses will not be queried for reports by `cf-hub`, even though they are in the last-seen database.

The lists may contain network addresses in CIDR notation or regular expressions to match the IP address. However, host names are currently not supported.

5.9.3 hub_schedule

Type: slist
Allowed input range: (arbitrary string)
Synopsis: The class schedule used by cf-hub for report collation
Example:

```
body hub control
{
hub_schedule => { "Min00", "Min30", "(Evening|Night).Min45_50" };
}
```

Notes:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

5.9.4 port

Type: int
Allowed input range: 1024,99999
Default value: 5308
Synopsis: Default port for contacting hub nodes
Example:

```
body hub control
{
port => "5308";
}

body server control
{
specialhost::
 port => "5308";

!specialhost::
 port => "5308";
}
```

Notes:

The standard or registered port number is tcp/5308. CFEngine does not presently use its registered udp port with the same number, but this could change in the future.

Changing the standard port number is not recommended practice. You should not do it without a good reason.

5.10 file control promises

```
body file control
{
namespace => "name1";
}
```

```
bundle agent private
{
....
}
```

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

This directive can be given multiple times within any file, outside of body and bundle definitions.

5.10.1 namespace

Type: string

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: Switch to a private namespace to protect current file from duplicate definitions

Example:

```
body file control
{
namespace => "name1";
}
```

For fuller examples see See Section 2.8 [Name spaces], page 28.

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

This directive can be given within any file, outside of body and bundle definitions, to change the namespace of subsequent bundles and bodies.

6 Bundles of common

```
bundle common globals
{
vars:

  "global_var" string => "value";

classes:

  "global_class" expression => "value";
}
```

Common bundles may only contain the promise types that are common to all bodies. Their main function is to define cross-component global definitions. Common bundles are observed by every agent, whereas the agent specific bundle types are ignored by components other than the intended recipient.

6.1 classes promises in '*'

Classes promises refer to the classification of different run-contexts. These are not related to object oriented classes, they are more like tag labels representing different properties of the environement.

The term classes was introduced before the widespread use of the term in Object Orientation, and has since stuck. Today, we use the words class and context interchangeably.

```
bundle common g
{
classes:

  "one" expression => "any";

  "client_network" expression => iprange("128.39.89.0/24");
}
```

Classes promises may be made in any bundle, i.e. in bundles pertaining to any agent. Classes that are defined in common bundles are global in scope, while classes in all other bundles are local.

6.1.1 and

Type: clist

Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`

Synopsis: Combine class sources with AND

Example:

```
classes:

  "compound_class" and => { classmatch("host[0-9].*"), "Monday", "Hr02" };
```

Notes:

If an expression contains a mixture of different object types that need to be ANDed together, this list form is more convenient than providing an expression. If all of the class expressions listed in the RHS match, then the class on the LHS is defined.

6.1.2 dist

Type: rlist

Allowed input range: `-9.99999E100,9.99999E100`

Synopsis: Generate a probabilistic class distribution (from strategies in cfengine 2)

Example:

```
classes:

  "my_dist"

    dist => { "10", "20", "40", "50" };
```

Notes:

Assign one generic class (always) and one additional class, randomly weighted on a probability distribution. The sum of 10+20+40+50 = 120 in the example above, so in generating a distribution, CFEngine picks a number between `1-120`. This will generate the following classes:

```
my_dist    (always)
my_dist_10 (10/120 of the time)
my_dist_20 (20/120 of the time)
my_dist_40 (40/120 of the time)
my_dist_50 (50/120 of the time)
```

This was previous called a '`strategy`' in CFEngine 2.

6.1.3 expression

Type: class
Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`
Synopsis: Evaluate string expression of classes in normal form
Example:

```
classes:

  "class_name" expression => "solaris|(linux.specialclass)";
  "has_toor"   expression => userexists("toor");
```

Notes:

A way of aliasing class combinations.

6.1.4 or

Type: clist
Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`
Synopsis: Combine class sources with inclusive OR
Example:

```
classes:

    "compound_test"

      or => { classmatch("linux_x86_64_2_6_22.*"), "suse_10_3" };
```

Notes:

A useful construction for writing expressions that contain special functions. The class in the LHS will be defined if any one (or more) of the class expressions in the RHS are true.

6.1.5 persistence

Type: int
Allowed input range: `0,99999999999`
Synopsis: Make the class persistent (cached) to avoid reevaluation, time in minutes
Example:

```
bundle common setclasses
{
classes:

  "cached_classes"
                or => { "any" },
       persistence => "1";

  "cached_class"
       expression => "any",
       persistence => "1";

}
```

Notes:

History: Was introduced in 3.4.0, Nova 2.2.0 (2012)

This feature can be used to avoid recomputing expensive classes calculations on each invocation. If a class discovered is essentially constant or only slowly varying (like a hostname or alias from a non-standard naming facility)

For example, to create a conditional inclusion of costly class definitions, put them into a separate bundle in a file 'classes.cf'.

```
# promises.cf

body common control
{
cached_classes::
  bundlesequence => { "test" };

!cached_classes::
  bundlesequence => {  "setclasses", "test" };

!cached_classes::
  inputs => { "classes.cf" };
}


bundle agent test
{
reports:

  !my_cached_class::
   "no cached class";

  my_cached_class::
    "cached class defined";
```

```
}
```

Then create 'classes.cf'

```
# classes.cf

bundle common setclasses
{
classes:

  "cached_classes"            # timer flag
         expression => "any",
        persistence => "480";

  "my_cached_class"
                or => { ...long list or heavy function... } ,
       persistence => "480";

}
```

6.1.6 not

Type: class

Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`

Synopsis: Evaluate the negation of string expression in normal form

Example:

```
classes:

   "others"  not => "linux|solaris";
   "no_toor" not => userexists("toor");
```

Notes:

This negates the effect of the promiser-pattern regular expression. The class in the LHS will only be defined if the class expression in the RHS is false.

6.1.7 select_class

Type: clist

Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`

Default value: random_selection

Synopsis: Select one of the named list of classes to define based on host identity

Example:

```
bundle common g
{
classes:
  "selection" select_class => { "one", "two" };

reports:
  one::
    "One was selected";
  two::
    "Two was selected";
  selection::
     "A selection was made";
}
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1.0 (2011)

This feature is somewhat like the `splayclass` function, but instead of selecting a class for a moment in time, it always chooses one class in the list – the same class each time for a given host. This allows hosts to be distributed across a controlled list of classes, e.g for load balancing purposes.

The class is chosen deterministically (not randomly) but it is not possible to say which host will end up in which class in advance – only that hosts will always end up in the same class every time.

6.1.8 `xor`

Type: clist

Allowed input range: `[a-zA-Z0-9_!&@@$|.()\[\]{}:]+`

Synopsis: Combine class sources with XOR

Example:

```
classes:

 "another_global" xor => { "any", "linux", "solaris"};
```

Notes:

Behaves as the XOR operation on class expressions. It can be used to define a class if exactly one of the class expressions on the RHS matches.

6.2 defaults promises in '*'

Defaults promises are related to variables. If a variable or paramter in a promise bundle is undefined, or its value is defined to be invalid, a default value can be promised instead.

A basic example, illustrating different ways to test for missing input. Note carefully that CFEngine does not use Perl semantics: i.e. undefined variables do not map to the empty string, they remain as variables for possible future expansion. Because of the convergence of CFEngine, some variables might in the strictest sense be defined but still contain unresolved variables, to handle this you will need to match the `$(abc)` form of the variables.

```
body common control
{
bundlesequence => { "main" };
}

bundle agent main
{
methods:

  "example"  usebundle => test("one","x","","$(four)");

}

bundle agent test(a,b,c,d)
{
defaults:

 "a" string => "default a", if_match_regex => "";
 "b" string => "default b", if_match_regex => "x";
 "c" string => "default c", if_match_regex => "";
 "d" string => "default d", if_match_regex => "\$\([a-zA-Z0-9_.]+\)";

reports:

   !nothing::

   "a = '$(a)', b = '$(b)', c = '$(c)' d = '$(d)'";
}
```

Another example:

```
bundle agent example
```

```
{
defaults:

  "X" string => "I am a default value";
  "Y" slist => { "I am a default list item 1", "I am a default list item 2" };

methods:

 "example" usebundle => mymethod("","bbb");

reports:

 !xyz::

   "The default value of X is $(X)";
   "The default value of Y is $(Y)";
}

###########################################################

bundle agent mymethod(a,b)

{
vars:

  "no_return" string => "ok"; # readfile("/dont/exist","123");

defaults:

  "a" string => "AAAAAAAAA",   if_match_regex => "";

  "b" string => "BBBBBBBBB",   if_match_regex => "";

  "no_return" string => "no such file";

reports:

  !xyz::

     "The value of a is $(a)";
     "The value of b is $(b)";

     "The value of no_return is $(no_return)";

}
```

This was introduced in CFEngine core 3.4.0. (2012)

6.2.1 if_match_regex

Type: string

Allowed input range: (arbitrary string)

Synopsis: If this regular expression matches the current value of the variable, replace it with default

Example:

```
bundle agent mymethod(a,b)
{
defaults:

  "a" string => "AAAAAAAAA",   if_match_regex => "";
  "b" string => "BBBBBBBBB",   if_match_regex => "";
}
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

If a parameter or variable is already defined in the current context and the value matches this regular expression, it will be deemed invalid and replaced with the default value.

6.2.2 string

Type: string

Allowed input range: (arbitrary string)

Synopsis: A scalar string

Example:

```
vars:

 "xxx"    string => "Some literal string...";

 "yyy"    string => readfile( "/home/mark/tmp/testfile" , "33" );
```

Notes:

In CFEngine previously lists were represented (as in the shell) using separted scalars, e.g. like the PATH variable. This design feature turned out to be an error of judgement which has resulted in

much trouble. This is no longer supported in CFEngine 3. By keeping lists an independent type many limitations have been removed.

6.2.3 slist

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of scalar strings

Example:

```
vars:

 "xxx"    slist  => {  "literal1",  "literal2" };

 "yyy"    slist  => {
                    readstringlist(
                                  "/home/mark/tmp/testlist",
                                  "#[a-zA-Z0-9 ]*",
                                  "[^a-zA-Z0-9]",
                                  15,
                                  4000
                                  )
                    };

 "zzz"    slist  => { readstringlist("/home/mark/tmp/testlist2","#[^\n]*",",",5,4000) };
```

Notes:

Some functions return slists (see Section 11.1 [Introduction to functions], page 411), and an slist may contain the values copied from another slist, rlist, or ilist (see Section 2.7.2 [List variable substitution and expansion], page 24, see Section 6.5.7 [policy in vars], page 166).

6.3 meta promises in '*'

Meta-data promises have no internal function. They are intended to be used to represent arbitrary information about promise bundles. Formally, meta promises are implemented as variables, and the values map to a variable context called *bundlename_meta*, and therefore the values can be used as variables and will appear in Enterprise variable reports.

```
bundle agent example
```

```
{
meta:

  "bundle_version" string => "1.2.3";
  "works_with_cfengine" string => "3.4.0";

reports:

 cfengine_3::

  "Not a local variable: $(bundle_version)";
  "Meta data (variable): $(example_meta.bundle_version)";

}
```

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

6.3.1 string

Type: string

Allowed input range: (arbitrary string)

Synopsis: A scalar string

Example:

```
vars:

 "xxx"    string => "Some literal string...";

 "yyy"    string => readfile( "/home/mark/tmp/testfile" , "33" );
```

Notes:

In CFEngine previously lists were represented (as in the shell) using separted scalars, e.g. like the PATH variable. This design feature turned out to be an error of judgement which has resulted in much trouble. This is no longer supported in CFEngine 3. By keeping lists an independent type many limitations have been removed.

6.3.2 slist

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of scalar strings

Example:

```
vars:

 "xxx"    slist  => {  "literal1",  "literal2" };

 "yyy"    slist  => {
                readstringlist(
                              "/home/mark/tmp/testlist",
                              "#[a-zA-Z0-9 ]*",
                              "[^a-zA-Z0-9]",
                              15,
                              4000
                              )
                };

 "zzz"    slist  => { readstringlist("/home/mark/tmp/testlist2","#[^\n]*",",",5,4000) };
```

Notes:

Some functions return slists (see Section 11.1 [Introduction to functions], page 411), and an slist may contain the values copied from another slist, rlist, or ilist (see Section 2.7.2 [List variable substitution and expansion], page 24, see Section 6.5.7 [policy in vars], page 166).

6.4 reports promises in '*'

Reports promises simply print messages. Outputting a message without qualfication can be a 'dangerous' operation. In a large installation it could unleash an avalanche of messaging. For that reason reports promises are not allowed to be in the class 'any'; they must be in a more specific class – this is not a fool-proof protection but it generally weeds out simple carelessness.

```
reports:

 "literal string or file refererence",

     printfile => printfile_body,
     ...;
```

Messages outputted from report promises are prefixed with the letter 'R' to distinguish them from other output, e.g. from commands.

In CFEngine 2 reporting was performed with alerts. The phrase alerts seems too strong and has been replaced by reports.

```
bundle agent report
{
reports:

  linux::

   "/etc/passwd except $(const.n)"

    # printfile => pr("/etc/passwd","5");

     showstate => { "otherprocs", "rootprocs" };

}
```

6.4.1 friend_pattern

Type: string
Allowed input range: (arbitrary string)
Synopsis: Regular expression to keep selected hosts from the friends report list
Example:

```
reports:

  linux::

   "Friend status report"

          lastseen => "0",
     friend_pattern => "host1|host2|.*\.domain\.tld";
```

Notes:

This regular expression should match hosts we want to exclude from friend reports.

6.4.2 intermittency

Type: real
Allowed input range: `0,1`
Default value: false

Synopsis: Real number threshold [0,1] of intermittency about current peers, report above
Example:

Notes:

6.4.3 lastseen

Type: int
Allowed input range: `0,99999999999`
Synopsis: Integer time threshold in hours since current peers were last seen, report absence
Example:

In control:

```
body agent control
{
lastseen => "false";
}
```

See also in reports:

```
reports:

  "Comment"

    lastseen => "10";
```

Notes:

In reports: after this time (hours) has passed, CFEngine will begin to warn about the host being overdue. After the `lastseenexpireafter` expiry time, hosts will be purged from this host's database (default is a week).

In control: determines whether CFEngine will records last seen intermittency profiles (reliability diagnostics) in '`WORKDIR/lastseen`'. This generates a separate file for each each host that connects to the current host. For central hubs this can result is a huge number of files.

6.4.4 printfile (body template)

Type: (ext body)

'`file_to_print`'

  Type: string

  Allowed input range: `"?(/.*)`

Synopsis: Path name to the file that is to be sent to standard output

Example:

```
body printfile example
{
file_to_print   => "/etc/motd";
number_of_lines => "10";
}
```

Notes:

Include part of a file in a report.

'number_of_lines'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Integer maximum number of lines to print from selected file

Example:

```
body printfile example
{
number_of_lines => "10";
}
```

Notes:

6.4.5 report_to_file

Type: string

Allowed input range: `"?(/.*)`

Synopsis: The path and filename to which output should be appended

Example:

```
bundle agent test
{
reports:
```

```
  linux::

   "$(sys.date),This is a report from $(sys.host)"

       report_to_file => "/tmp/test_log";
}
```

Notes:

Append the output of the report to the named file instead of standard output. If the file cannot be opened for writing then the report defaults to the standard output.

6.4.6 bundle_return_value_index

Type: string

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: The promiser is to be interpreted as a literal value that the caller can accept as a result for this bundle, i.e. a return value with array index defined by this attribute.

Example:

```
body common control
{
bundlesequence => { "test" };
}


bundle agent test
{
methods:

   "any" usebundle => child,
    useresult => "my_return_var";


reports:

  cfengine_3::

    "My return was: \"$(my_return_var[1])\" and \"$(my_return_var[2])\"";

}

bundle agent child
{
reports:
```

```
 cfengine_3::

   # Map these indices into the useresult namespace

   "this is a return value"
      bundle_return_value_index => "1";

   "this is another return value"
      bundle_return_value_index => "2";

}
```

Notes:

History: Was introduced in 3.4.0 (2012)

It is currently believed that only scalar return values make sense, hence return values are limited to scalars.

6.4.7 showstate

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of services about which status reports should be reported to standard output

Example:

```
reports:
  cfengine::

    "Comment"

      showstate => {"www_in", "ssh_out", "otherprocs" };
```

Notes:

The basic list of services is:

'users' Users logged in

'rootprocs'
Privileged system processes

'otherprocs'
Non-privileged process

'diskfree' Free disk on / partition

'loadavg' % kernel load utilization

'netbiosns_in'
netbios name lookups (in)

'netbiosns_out'
netbios name lookups (out)

'netbiosdgm_in'
netbios name datagrams (in)

'netbiosdgm_out'
netbios name datagrams (out)

'netbiosssn_in'
netbios name sessions (in)

'netbiosssn_out'
netbios name sessions (out)

'irc_in' IRC connections (in)

'irc_out' IRC connections (out)

'cfengine_in'
CFEngine connections (in)

'cfengine_out'
CFEngine connections (out)

'nfsd_in' nfs connections (in)

'nfsd_out' nfs connections (out)

'smtp_in' smtp connections (in)

'smtp_out' smtp connections (out)

'www_in' www connections (in)

'www_out' www connections (out)

'ftp_in' ftp connections (in)

'ftp_out' ftp connections (out)

'ssh_in' ssh connections (in)

'ssh_out' ssh connections (out)

'wwws_in' wwws connections (in)

'wwws_out' wwws connections (out)

'icmp_in' ICMP packets (in)

'icmp_out' ICMP packets (out)

'udp_in' UDP dgrams (in)

'udp_out' UDP dgrams (out)

'dns_in' DNS requests (in)

'dns_out' DNS requests (out)

'tcpsyn_in'
TCP sessions (in)

'tcpsyn_out'
TCP sessions (out)

'tcpack_in'
TCP acks (in)

'tcpack_out'
TCP acks (out)

'tcpfin_in'
TCP finish (in)

'tcpfin_out'
TCP finish (out)

'tcpmisc_in'
TCP misc (in)

'tcpmisc_out'
TCP misc (out)

'webaccess'
Webserver hits

'weberrors'
Webserver errors

'syslog' New log entries (Syslog)

'messages' New log entries (messages)

'temp0' CPU Temperature 0

'temp1' CPU Temperature 1

'temp2' CPU Temperature 2

'temp3' CPU Temperature 3

'cpu' %CPU utilization (all)

'cpu0' %CPU utilization 0

'cpu1' %CPU utilization 1

'cpu2' %CPU utilization 2

'cpu3' %CPU utilization 3

6.5 vars promises in '*'

Variables in CFEngine are defined as promises that an identifier represents a particular value. Variables in CFEngine are dynamically typed as strings, integers and real numbers. Lists of these types are also possible.

CFEngine supports a few operations on arrays. Arrays are simply a naming convention for scalar variables, using square brackets '[]' to enclose an arbitrary key. Arrays are thus `associative`.

```
bundle agent example

{
vars:

  "scalar1" string => "SCALAR 1";
  "list1" slist => { "LIST1_1", "LIST1_2" } ;
  "array[1]" string => "ARRAY 1";
  "array[2]" string => "ARRAY 2";

  "i" slist => getindices("array");

reports:

  cfengine_3::

    "Scalar $(scalar1)";
    "List $(list1)";
    "Array $(array[$(i)])";
}
```

6.5.1 string

Type: string

Allowed input range: (arbitrary string)

Synopsis: A scalar string

Example:

```
vars:
```

```
 "xxx"    string => "Some literal string...";

 "yyy"    string => readfile( "/home/mark/tmp/testfile" , "33" );
```

Notes:

In CFEngine previously lists were represented (as in the shell) using separted scalars, e.g. like the PATH variable. This design feature turned out to be an error of judgement which has resulted in much trouble. This is no longer supported in CFEngine 3. By keeping lists an independent type many limitations have been removed.

6.5.2 int

Type: int
Allowed input range: -99999999999,9999999999
Synopsis: A scalar integer
Example:

```
vars:

 "scalar" int    => "16k";

 "ran"    int    => randomint(4,88);

 "dim_array" int =>  readstringarray("array_name","/etc/passwd","#[^\n]*",":",10,4000);
```

Notes:

Int variables are strings that are expected to be used as integer numbers. The typing in CFEngine is dynamic, so the variable types are interchangable, but when you declare a variable to be type `int`, CFEngine verifies that the value you assign to it looks like an integer (e.g., '3', '-17', '16K', etc).

Integer values may use suffices 'k', 'K', 'm', 'M', etc., but must only have an integer numeric part (so '1.5M' is not allowed).

'k' The value multipled by 1000.

'K' The value multipled by 1024.

'm' The value multipled by 1000 * 1000.

'M' The value multipled by 1024 * 1024.

'g' The value multipled by 1000 * 1000 * 1000.

'G' The value multipled by 1024 * 1024 * 1024.

'%' A percentage between 1 and 100 - mainly for use in a storage context.

The value 'inf' may also be used to represent an unlimited positive value.

6.5.3 real

Type: real

Allowed input range: `-9.99999E100,9.99999E100`

Synopsis: A scalar real number

Example:

```
vars:

 "scalar" real   => "0.5";
```

Notes:

Real variables are strings that are expected to be used as real numbers. The typing in CFEngine is dynamic, so the variable types are interchangable, but when you declare a variable to be type `real`, CFEngine verifies that the value you assign to it looks like a real number (e.g., '3', '3.1415', '.17', '6.02e23', '-9.21e-17', etc).

Real numbers are not used in many places in CFEngine, but they are useful for representing probabilties and performance data.

6.5.4 slist

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of scalar strings

Example:

```
vars:

 "xxx"    slist  => {  "literal1",  "literal2" };

 "yyy"    slist  => {
                    readstringlist(
                                  "/home/mark/tmp/testlist",
                                  "#[a-zA-Z0-9 ]*",
                                  "[^a-zA-Z0-9]",
                                  15,
                                  4000
                                  )
                    };

 "zzz"    slist  => { readstringlist("/home/mark/tmp/testlist2","#[^\n]*",",",5,4000) };
```

Notes:

Some functions return slists (see Section 11.1 [Introduction to functions], page 411), and an slist may contain the values copied from another slist, rlist, or ilist (see Section 2.7.2 [List variable substitution and expansion], page 24, see Section 6.5.7 [policy in vars], page 166).

6.5.5 ilist

Type: ilist

Allowed input range: -99999999999,9999999999

Synopsis: A list of integers

Example:

```
vars:

  "variable_id"

       ilist => { "10", "11", "12" };
```

Notes:

Integer lists are lists of strings that are expected to be treated as integers. The typing in CFEngine is dynamic, so the variable types are interchangable, but when you declare a variable to be type ilist, CFEngine verifies that each value you assign to it looks like an integer (e.g., '3', '-17', '16K', etc).

Some functions return ilists (see Section 11.1 [Introduction to functions], page 411), and an ilist may contain the values copied from another slist, rlist, or ilist (see Section 2.7.2 [List variable substitution and expansion], page 24, see Section 6.5.7 [policy in vars], page 166).

6.5.6 rlist

Type: rlist

Allowed input range: -9.99999E100,9.99999E100

Synopsis: A list of real numbers

Example:

```
vars:

  "varid" rlist => { "0.1", "0.2", "0.3" };
```

Notes:

Real lists are lists of strings that are expected to be used as real numbers. The typing in CFEngine is

dynamic, so the variable types are interchangable, but when you declare a variable to be type `rlist`, CFEngine verifies that each value you assign to it looks like a real number (e.g., '3', '3.1415', '.17', '6.02e23', '-9.21e-17', etc).

Some functions return `rlists` (see Section 11.1 [Introduction to functions], page 411), and an `rlist` may contain the values copied from another `slist`, `rlist`, or `ilist` (see Section 2.7.2 [List variable substitution and expansion], page 24, see Section 6.5.7 [policy in vars], page 166).

6.5.7 policy

Type: (menu option)
Allowed input range:

```
free
overridable
constant
ifdefined
```

Synopsis: The policy for (dis)allowing (re)definition of variables
Example:

```
vars:

  "varid" string => "value...",
          policy => "constant";
```

Notes:

Variables can either be allowed to change their value dynamically (be redefined) or they can be constant. The use of private variable spaces in CFEngine 3 makes it unlikely that variable redefinition would be necessary in CFEngine 3.

The value `constant` indicates that the variable value may not be changed. The values `free` and `overridable` are synonymous, and indicated the the variable's value may be changed.

The value `ifdefined` applies only to lists and implies that unexpanded or undefined lists are dropped. The default behaviour is otherwise to retain this value as an indicator of the failure to quench the variable reference, e.g.

```
    "one" slist => { "1", "2", "3" };

   "list" slist => { "@(one)", @(two) },

            policy => "ifdefined";
```

would result in '@(list)' being the same as '@(one)', and the reference to '@(two)' would disappear. This is useful for combining lists, 'inheritance-style' where one can extend a base with special cases if they are defined.

Default value:

```
policy => constant
```

6.6 * promises

Whereas most promise types are specific to a particular kind of interpretation that requires a typed interpreter (the bundle type), a number of promises can be made in any kind of bundle since they are of a generic input/output nature. These are `vars`, `classes`, and `reports` promises. The specific promise attributes are listed below.

6.6.1 `action` (body template)

Type: (ext body)

'`action_policy`'

Type: (menu option)

Allowed input range:

```
fix
warn
nop
```

Synopsis: Whether to repair or report about non-kept promises

Example:

The following example shows a simple use of transaction control, causing the promise to be verified as a separate background process.

```
body action background

{
action_policy => "warn";
}
```

Notes:

The `action` settings allow general transaction control to be implemented on promise verification. Action bodies place limits on how often to verify the promise and what classes to raise in the case that the promise can or cannot be kept.

Note that actions can be added to sub-bundles like methods and editing bundles, and that promises within these do not inherit action settings at higher levels. Thus, in the following example there are two levels of action setting:

```
#######################################################
#
# Warn if line matched
#
#######################################################

body common control

{
bundlesequence  => { "testbundle" };
}

#######################################################

bundle agent testbundle

{
files:

  "/var/cfengine/inputs/.*"

       edit_line => DeleteLinesMatching(".*cfenvd.*"),
       action => WarnOnly;
}

#######################################################

bundle edit_line DeleteLinesMatching(regex)
  {
  delete_lines:

    "$(regex)" action => WarnOnly;

  }

#######################################################

body action WarnOnly
{
action_policy => "warn";
}
```

The `action` setting for the `files` promise means that file edits will not be committed to disk, only warned about. This is a master-level promise that overrides anything that happens during the editing. The `action` setting for the edit bundle means that the internal memory modelling of the file will only warn about changes rather than committing them

to the memory model. This makes little difference to the end result, but it means that CFEngine will report

```
Need to delete line - ... - but only a warning was promised
```

Instead of

```
Deleting the prpomised line ...
Need to save file - but only a warning was promised
```

In either case, no changes will be made to the disk, but the messages given by `cf-agent` will differ.

'`ifelapsed`'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Number of minutes before next allowed assessment of promise

Default value: control body value

Example:

```
#local

body action example
{
ifelapsed   => "120";      # 2 hours
expireafter => "240";      # 4 hours
}

# global

body agent control
{
ifelapsed   => "180";      # 3 hours
}
```

Notes:

This overrides the global settings. Promises which take a long time to verify should usually be protected with a long value for this parameter. This serves as a resource 'spam' protection. A CFEngine check could easily run every 5 minutes provided resource intensive operations are not performed on every run. Using time classes like `Hr12` etc., is one part of this strategy; using `ifelapsed` is another which is not tied to a specific time.

'`expireafter`'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Number of minutes before a repair action is interrupted and retried

Default value: control body value

Example:

```
body action example
{
ifelapsed   => "120";      # 2 hours
expireafter => "240";      # 4 hours
}
```

Notes:

The locking time after which CFEngine will attempt to kill and restart its attempt to keep a promise.

'log_string'

Type: string

Allowed input range: (arbitrary string)

Synopsis: A message to be written to the log when a promise verification leads to a repair

Example:

```
promise-type:

 "promiser"

   attr => "value",
   action => log_me("checked $(this.promiser) in promise $(this.handle)");

# ..

body action log_me(s)
{
log_string => "$(s)";
}
```

Notes:

The log_string works together with log_repair, log_kept etc, to define a string for logging to one of the named files depending on promise outcome, or to standard output of the log file is stipulared as 'stdout'. Log strings on standard output are denoted by an 'L:' prefix.

Note that log_string does not interact with log_level, which is about regular system output messages.

Hint: the promise handle '$(this.handle)' can be a useful referent in a log message, indicating the origin of the message. In CFEngine Nova and above, every promise has a default handle (which is based o the filename and line number - specifying your own handle will probably be more mnemonic)..

'log_level'

Type: (menu option)

Allowed input range:

```
inform
verbose
error
log
```

Synopsis: The reporting level sent to syslog

Example:

```
body action example
{
log_level => "inform";
}
```

Notes:

Use this as an alternative to auditing to use the syslog mechanism to centralize or manage messaging from CFEngine. A backup of these messages will still be kept in 'WORKDIR/outputs' if you are using cf-execd.

On native Windows version of CFEngine (Nova or above), using 'verbose' will include a message when the promise is kept or repaired in the event log.

'log_kept' **Type**: string

Allowed input range: `stdout|udp_syslog|("?[a-zA-Z]:\\.*)|(/.*)`

Synopsis: This should be filename of a file to which log_string will be saved, if undefined it goes to the system logger

Example:

```
body action logme(x)
{
log_kept => "/tmp/private_keptlog.log";
```

```
log_failed => "/tmp/private_faillog.log";
log_repaired => "/tmp/private_replog.log";
log_string => "$(sys.date) $(x) promise status";
}
```

Notes:

If this option is specified together with `log_string`, the current promise will log promise-kept status using the log string to this named file. If these log names are absent, the default logging destination for the log string is syslog, but only for non-kept promises. Only the `log_string` is affected by this setting. Other messages destined for logging are sent to syslog.

It is intended that named file logs should be different for the three cases: promise kept, promise not kept and promise repaired.

This string should be the full path to a text file which will contain the log, of one of the following special values:

'`stdout`' Send the log message to the standard output, prefixed with an '`L:`' to indicate a log message.

'`udp_syslog`'
Attempt to connect to the `syslog_server` defined in '`body common control`' and log the message there, assuming the server is configured to receive the request.

'`log_priority`'

Type: (menu option)

Allowed input range:

```
emergency
alert
critical
error
warning
notice
info
debug
```

Synopsis: The priority level of the log message, as interpreted by a syslog server

Example:

```
body action low_priority
{
log_priority => "info";
}
```

Notes:

This determines the importance of messages from CFEngine.

'log_repaired'

Type: string

Allowed input range: `stdout|udp_syslog|("?[a-zA-Z]:\\.*)|(/.*)`

Synopsis: This should be filename of a file to which log_string will be saved, if undefined it goes to the system logger

Example:

```
bundle agent test
{
vars:

  "software" slist => { "/root/xyz", "/tmp/xyz" };

files:

  "$(software)"

    create => "true",
     action => logme("$(software)");

}

body action logme(x)
{
log_kept => "/tmp/private_keptlog.log";
log_failed => "/tmp/private_faillog.log";
log_repaired => "/tmp/private_replog.log";
log_string => "$(sys.date) $(x) promise status";
}

body action immediate_syslog(x)
{
log_repaired => "udp_syslog"; # Nova and above
log_string => "CFEngine repaired promise $(this.handle) - $(x)";
}
```

Notes:

This may be the name of a log to which the `log_string` is written if a promise is repaired. It should be the full path to a text file which will contain the log, of one of the following special values:

'stdout' Send the log message to the standard output, prefixed with an 'L:' to indicate a log message.

'udp_syslog'
Attempt to connect to the `syslog_server` defined in 'body common control' and log the message there, assuming the server is configured to receive the request.

'log_failed'

Type: string

Allowed input range: `stdout|udp_syslog|("?[a-zA-Z]:\\.*)|(/.*)`

Synopsis: This should be filename of a file to which log_string will be saved, if undefined it goes to the system logger

Example:

```
bundle agent test
{
vars:

  "software" slist => { "/root/xyz", "/tmp/xyz" };

files:

  "$(software)"

    create => "true",
     action => logme("$(software)");

}


body action logme(x)
{
log_kept => "/tmp/private_keptlog.log";
log_failed => "/tmp/private_faillog.log";
log_repaired => "/tmp/private_replog.log";
log_string => "$(sys.date) $(x) promise status";
}
```

Notes:

If this option is specified together with log_string, the current promise will log promise-kept status using the log string to this named file. If these log names are absent, the default logging destination for the log string is syslog, but only for non-kept promises. Only the log_string is affected by this setting. Other messages destined for logging are sent to syslog.

It is intended that named file logs should be different for the three cases: promise kept, promise not kept and promise repaired. This string should be the full path to a text file which will contain the log, of one of the following special values:

'stdout' Send the log message to the standard output, prefixed with an 'L:' to indicate a log message.

'udp_syslog'
Attempt to connect to the syslog_server defined in 'body common control' and log the message there, assuming the server is configured to receive the request.

'value_kept'

Type: real

Allowed input range: -9.99999E100,9.99999E100

Synopsis: A real number value attributed to keeping this promise

Example:

```
body action mydef
{
value_kept     => "4.5";   # this promise is worth 4.5 dollars per hour
value_repaired => "2.5";   # fixing this promise is worth 2.5 dollars per hour
value_notkept  => "-10.0"; # not keeping this promise costs is 10 dollars per hour
ifelapsed       => "60";   # one hour
}
```

Notes:

If nothing is specified, the default value is +1.0. However, nothing is logged unless the agent control body switched on 'track_value => "true"'.

'value_repaired'

Type: real

Allowed input range: -9.99999E100,9.99999E100

Synopsis: A real number value attributed to reparing this promise

Example:

```
body action mydef
{
value_kept     => "4.5";   # this promise is worth 4.5 dollars per hour
value_repaired => "2.5";   # fixing this promise is worth 2.5 dollars per hour
value_notkept  => "-10.0"; # not keeping this promise costs is 10 dollars per hour
ifelapsed       => "60";   # one hour
}
```

Notes:

If nothing is specified, the default value is 0.5. However, nothing is logged unless the agent control body switched on ‘track_value => "true"’.

‘value_notkept’

Type: real

Allowed input range: -9.99999E100,9.99999E100

Synopsis: A real number value (possibly negative) attributed to not keeping this promise

Example:

```
body action mydef
{
value_kept     => "4.5";   # this promise is worth 4.5 dollars per hour
value_repaired => "2.5";   # fixing this promise is worth 2.5 dollars per hour
value_notkept  => "-10.0"; # not keeping this promise costs is 10 dollars per hour
ifelapsed       => "60";   # one hour
}
```

Notes:

If nothing is specified, the default value is -1.0. However, nothing is logged unless the agent control body switched on ‘track_value => "true"’.

‘audit’ **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false switch for detailed audit records of this promise

Default value: false

Example:

```
body action example
{
# ...

audit => "true";
}
```

Notes:

If this is set, CFEngine will perform auditing on this specific promise. This means that all details surrounding the verification of the current promise will be recorded in the audit database. The database may be inspected with `cf-report`, or `cfshow` in CFEngine 2.

'background'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false switch for parallelizing the promise repair

Default value: false

Example:

```
body action example
{
background => "true";
}
```

Notes:

If possible, perform the verification of the current promise in the background. This is advantageous only if the verification might take a significant amount of time, e.g. in remote copying of filesystem/disk scans.

On the windows version of CFEngine Nova, this can be useful if we don't want to wait for a particular command to finish execution before checking the next promise. This is particular for the windows platform because there is no way that a program can start itself in the background here (i.e. fork off a child process). However, file operations can not be performed in the background on windows.

'report_level'

Type: (menu option)

Allowed input range:

```
inform
verbose
error
log
```

Synopsis: The reporting level for standard output for this promise

Default value: none

Example:

```
body action example
{
report_level => "verbose";
}
```

Notes:

cf-agent can be run in verbose mode (-v), inform mode (-I) and just print errors (no arguments). This attribute allows to set these three output levels on a per promise basis, allowing the promise to be more verbose than the global setting (but not less).

In CFEngine 2 one would say 'inform=true' or 'syslog=true', etc. This replaces these levels since they act as encapsulating super-sets.

'measurement_class'

Type: string

Allowed input range: (arbitrary string)

Synopsis: If set performance will be measured and recorded under this identifier

Example:

```
body action measure
{
measurement_class => "$(this.promiser) long job scan of /usr";
}
```

Notes:

By setting this string you switch on performance measurement for the current promise, and also give the measurement a name. The identifier forms a partial identity for optional performance scanning of promises of the form:

ID:promise-type:promiser.

These can be seen identifying using `cf-report`, e.g. in the generated file '`performance.html`'.

6.6.2 `classes` (body template)

Type: (ext body)

'`promise_repaired`'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be defined globally

Example:

```
body classes example
{
promise_repaired => { "change_happened" };
}
```

Notes:

If a promise is 'repaired' it means that a corrective action had to be taken to keep the promise.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

Important: complex promises, e.g. `files` promises that set multiple parameters on a file simultaneously can report misleadingly. The classes for different parts of a promise are not separable. Thus, if you promise to create and file and change its permissions, when the file exists with incorrect permissions, `cf-agent` will report that the '`promise_kept`' for the file existence, but '`promise_repaired`' for the permissions. If you need separate reports, you should code two separate promises rather than 'overloading' a single one.

'repair_failed'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be defined globally

Example:

```
body classes example
{
repair_failed => { "unknown_error" };
}
```

Notes:

A promise could not be repaired because the corrective action failed for some reason.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

'repair_denied'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be defined globally

Example:

```
body classes example
{
repair_denied => { "permission_failure" };
}
```

Notes:

A promise could not be kept because access to a key resource was denied.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

'repair_timeout'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be defined globally

Example:

```
body classes example
{
repair_timeout => { "too_slow", "did_not_wait" };
}
```

Notes:

A promise maintenance repair timed-out waiting for some dependent resource.

'promise_kept'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be defined globally

Example:

```
body classes example
{
promise_kept => { "success", "kaplah" };
}
```

Notes:

This class is set if no action was necessary by `cf-agent` because the promise concerned was already kept without further action required.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

Important: complex promises, e.g. `files` promises that set multiple parameters on a file simultaneously can report misleadingly. The classes for different parts of a promise are not separable. Thus, if you promise to create and file and change its permissions, when the file exists with incorrect permissions, `cf-agent` will report that the '`promise_kept`' for the file existence, but '`promise_repaired`' for the permissions. If you need separate reports, you should code two separate promises rather than 'overloading' a single one.

'cancel_kept'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be cancelled if the promise is kept

Example:

```
body classes example
{
cancel_kept => { "success", "kaplah" };
}
```

Notes:

If the promise was already kept and nothing was done, cancel (undefine) any of the listed classes so that they are no longer defined.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

History: This attribute was introduced in CFEngine version 3.0.4 (2010)

'cancel_repaired'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be cancelled if the promise is repaired

Example:

```
body classes example
{
cancel_repaired => { "change_happened" };
}
```

Notes:

If the promise was repaired and changes were made to the system, cancel (undefine) any of the listed classes so that they are no longer defined.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

History: This attribute was introduced in CFEngine version 3.0.4 (2010)

'cancel_notkept'

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of classes to be cancelled if the promise is not kept for any reason

Example:

```
body classes example
{
cancel_notkept => { "failure" };
}
```

Notes:

If the promise was not kept but nothing could be done, cancel (undefine) any of the listed classes so that they are no longer defined.

Note that any strings passed to this list are automatically canonified, so it is unecessary to call a canonify function on such inputs.

History: This attribute was introduced in CFEngine version 3.0.4 (2010)

'kept_returncodes'

Type: slist

Allowed input range: `[-0-9_$(){}\[\].]+`

Synopsis: A list of return codes indicating a kept command-related promise

Example:

```
bundle agent cmdtest
{
commands:
  "/bin/false"
   classes => example;

reports:
waskept::
  "The command-promise was kept!";
}

body classes example
{
kept_returncodes => { "0", "1" };
promise_kept => { "waskept" };
}
```

Notes:

A list of integer return codes indicating that a command-related promise has been kept. This can in turn be used to define classes using the `promise_kept` attribute, or merely alter the total compliance statistics.

Currently, the attribute has impact on the following command-related promises.

- All promises of type `commands:`
- `files`-promises containing a `transformer`-attribute
- The package manager change command in `packages`-promises (e.g. the command for add, remove, etc.)

If none of the attributes `kept_returncodes`, `repaired_returncodes`, or `failed_returncodes` are set, the default is to consider a return code zero as promise repaired, and nonzero as promise failed.

Note that the return codes may overlap, so multiple classes may be set from one return code. In Unix systems the possible return codes are usually in the range from 0 to 255.

History: Was introduced in version 3.1.3, Nova 2.0.2 (2010)

'`repaired_returncodes`'

Type: slist

Allowed input range: `[-0-9_$(){}\[\].]+`

Synopsis: A list of return codes indicating a repaired command-related promise

Example:

```
bundle agent cmdtest
{
commands:
  "/bin/false"
   classes => example;

reports:
wasrepaired::
  "The command-promise got repaired!";
}

body classes example
{
repaired_returncodes => { "0", "1" };
promise_repaired => { "wasrepaired" };
}
```

Notes:

A list of integer return codes indicating that a command-related promise has been repaired. This can in turn be used to define classes using the `promise_repaired` attribute, or merely alter the total compliance statistics.

Currently, the attribute has impact on the following command-related promises.

- All promises of type `commands:`
- `files`-promises containing a `transformer`-attribute
- The package manager change command in `packages`-promises (e.g. the command for add, remove, etc.)

If none of the attributes `kept_returncodes`, `repaired_returncodes`, or `failed_returncodes` are set, the default is to consider a return code zero as promise repaired, and nonzero as promise failed.

Note that the return codes may overlap, so multiple classes may be set from one return code. In Unix systems the possible return codes are usually in the range from 0 to 255.

History: Was introduced in version 3.1.3, Nova 2.0.2 (2010)

'failed_returncodes'

Type: slist

Allowed input range: `[-0-9_$(){}\[\].]+`

Synopsis: A list of return codes indicating a failed command-related promise

Example:

```
body common control
{
bundlesequence => { "cmdtest" };
}

bundle agent cmdtest
{
files:
"/tmp/test"
  copy_from => copy("/etc/passwd");


"/tmp/test"
  classes => example,
  transformer => "/bin/grep -q lkajfo999999 $(this.promiser)";

reports:
wasfailed::
  "The files-promise failed!";
}

body classes example
{
failed_returncodes => { "1" };
repair_failed => { "wasfailed" };
}
```

```
body copy_from copy(file)
{
source => "$(file)";
}
```

Notes:

A list of integer return codes indicating that a command-related promise has failed. This can in turn be used to define classes using the `promise_repaired` attribute, or merely alter the total compliance statistics.

Currently, the attribute has impact on the following command-related promises.

- All promises of type `commands:`
- `files`-promises containing a `transformer`-attribute
- The package manager change command in `packages`-promises (e.g. the command for add, remove, etc.)

If none of the attributes `kept_returncodes`, `repaired_returncodes`, or `failed_returncodes` are set, the default is to consider a return code zero as promise repaired, and nonzero as promise failed.

Note that the return codes may overlap, so multiple classes may be set from one return code. In Unix systems the possible return codes are usually in the range from 0 to 255.

History: Was introduced in version 3.1.3, Nova 2.0.2 (2010)

'persist_time'

Type: int

Allowed input range: `0,99999999999`

Synopsis: A number of minutes the specified classes should remain active

Example:

```
body classes example
{
persist_time => "10";
}
```

Notes:

By default classes are ephemeral entities that disappear when `cf-agent` terminates. By setting a persistence time, they can last even when the agent is not running.

'timer_policy'

Type: (menu option)

Allowed input range:

```
absolute
reset
```

Synopsis: Whether a persistent class restarts its counter when rediscovered

Default value: reset

Example:

```
body classes example
{
timer_policy => "reset";
}
```

Notes:

The in most cases resetting a timer will give a more honest appraisal of which classes are currently important, but if we want to activate a response of limited duration as a rare event then an asbolute time limit is useful.

6.6.3 comment

Type: string

Allowed input range: (arbitrary string)

Synopsis: A comment about this promise's real intention that follows through the program

Example:

```
comment => "This comment follows the data for reference ...",
```

Notes:

Comments written in code follow the program, they are not merely discarded. They appear in reports and error messages.

6.6.4 depends_on

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of promise handles that this promise builds on or depends on somehow (for knowledge management)

Example:

```
body common control
{
bundlesequence => { "one"  };
}

bundle agent one
{
reports:

 cfengine_3::

   "two"
     depends_on => { "handle_one" };

   "one"
     handle => "handle_one";

}
```

Notes:

This is a list of promise handles for whom this promise is a promisee. In other words, we acknowledge that this promise will be affected by the list of promises whose handles are specified.

It has the effect of partially ordering promises.

As of version 3.4.0, promise this feature is active and may be considered a short-hand for setting classes. If one promise depends on a list of others, it will not be verified unless the dependent promises have already been verified and kept: i.e. as long as the dependent promises are either kept or repaired the dependee can be verified.

Handles in other namespaces may be referred to by *namespace:handle*.

6.6.5 `handle`

Type: string

Allowed input range: (arbitrary string)

Synopsis: A unique id-tag string for referring to this as a promisee elsewhere

Example:

```
access:

  "/source"

    handle  => "update_rule",
```

```
    admit   => { "127.0.0.1" };
```

Notes:

A promise handle is like a 'goto' label. It allows you to refer to a promise as the promisee of `depends_on` client of another promise. Handles are essential for mapping dependencies and performing impact analyses. In Enterprise versions of CFEngine, promise handles can also be used in `outputs` promises, See Section 7.21 [outputs in agent promises], page 318.

Handles may consist of regular indentifier characters. CFEngine automatically 'canonifies' the names of handles to conform to this standard. Caution: if the handle name is based on a variable, and the variable fails to expand, the handle will be based on the name of the variable rather than its content.

6.6.6 ifvarclass

Type: string
Allowed input range: (arbitrary string)
Synopsis: Extended classes ANDed with context
Example:

The generic example has the form:

```
     promise-type:

       "promiser"

         ifvarclass => "$(program)_running|($(program)_notfound&Hr12)";
```

A specific example would be:

```
bundle agent example

{
commands:

 any::

    "/bin/echo This is linux"

       ifvarclass => "linux";


    "/bin/echo This is solaris"

       ifvarclass => "solaris";
```

```
}
```

Notes:

This is an additional class expression that will be evaluated after the '*class*::' classes have selected promises. It is provided in order to enable a channel between variables and classes. The result is thus the logical AND of the ordinary classes and the variable classes.

This function is provided so that one can form expressions that link variables and classes, e.g.

```
# Check that all components are running

vars:

  "component" slist => { "cf-monitord", "cf-serverd" };

processes:

  "$(component)" restart_class => canonify("start_$(component)");

commands:

   "/var/cfengine/bin/$(component)"

       ifvarclass => canonify("start_$(component)");
```

Notice that the function `canonify()` is provided to convert a general variable input into a string composed only of legal characters, using the same algorithm that CFEngine uses.

6.6.7 meta

Type: slist

Allowed input range: (arbitrary string)

Synopsis: User-data associated with policy, e.g. key=value strings

Example:

```
files:

  "/etc/special_file"

    comment => "Special file is a requirement. Talk to Fred X.",
    create => "true",

    meta => { "owner=John",  "version=2.0" };
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

Attending to knowledge management, it is sometimes convenient to attach meta-data of a more technical nature to policy. It may be used for arbitrary key=value strings for example.

7 Bundles of agent

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.1 commands promises in 'agent'

```
commands:

  "/path/to/command args"

             args => "more args",
             contain => contain_body,
             module => true/false;
```

Command *containment* allows you to make a 'sandbox' around a command, to run it as a non-privileged user inside an isolated directory tree. CFEngine `modules` are commands that support a simple

protocol (see below) in order to set additional variables and classes on execution from user defined code. Modules are intended for use as system probes rather than additional configuration promises.

In CFEngine 3 commands and processes have been separated cleanly. Restarting of processes must be coded as a separate command. This stricter type separation will allow more careful conflict analysis to be carried out.

Output from commands executed here is quoted inline, but prefixed with the letter 'Q' to distinguish it from other output, e.g. from `reports` (which is prefixed with the letter 'R').

It is possible to set classes based on the return code of a commands-promise in a very flexible way. See the `kept_returncodes`, `repaired_returncodes` and `failed_returncodes` attributes.

Commands were called `shellcommands` in CFEngine 2.

NOTE: a common mistake in using CFEngine is to embed many shell commands instead of using the built-in functionality. Use of CFEngine internals is preferred as it assures convergence and proper integrated checking. Extensive use of shell commands will make a CFEngine execution very heavy-weight like other management systems. To minimize the system cost of execution, always use CFEngine internals.

```
bundle agent example

{
commands:

  "/bin/sleep 10"
     action  => background;

  "/bin/sleep"
     args => "20",
     action  => background;

}
```

NOTE: when referring to executables whose paths contain spaces, you should quote the entire program string separately so that CFEngine knows the name of the executable file. e.g.

```
commands:

 windows::

  "\"c:\Program Files\my name with space\" arg1 arg2";

 linux::

  "\"/usr/bin/funny command name\" -a -b -c";
```

7.1.1 args

Type: string

Allowed input range: (arbitrary string)

Synopsis: Alternative string of arguments for the command (concatenated with promiser string)

Example:

```
commands:

  "/bin/echo one"

   args => "two three";
```

Notes:

Sometimes it is convenient to separate the arguments to a command from the command itself. The final arguments are the concatenation with one space. So in the example above the command would be

```
 /bin/echo one two three
```

7.1.2 contain (body template)

Type: (ext body)

'useshell' **Type**: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false embed the command in a shell environment

Default value: false

Example:

```
body contain example
{
useshell => "true";
```

```
}
```

Notes:

The default is to *not* use a shell when executing commands. Use of a shell has both resource and security consequences. A shell consumes an extra process and inherits environment variables, reads commands from files and performs other actions beyond the control of CFEngine. If one does not need shell functionality such as piping through multiple commands then it is best to manage without it. In the windows version of CFEngine Nova, the command is run in the the "Command Prompt" if useshell is true.

'umask' **Type**: (menu option)

Allowed input range:

```
0
77
22
27
72
077
022
027
072
```

Synopsis: The umask value for the child process

Example:

```
body contain example
{
umask => "077";
}
```

Notes:

Sets the internal umask for the process. Default value for the mask is '077'. On windows, umask is not supported and is thus ignored by windows versions of CFEngine.

'exec_owner'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The user name or id under which to run the process

Example:

```
body contain example
{
exec_owner => "mysql_user";
}
```

Notes:

This is part of the restriction of privilege for child processes when running cf-agent as the root user, or a user with privileges.

Windows requires the clear text password for the user account to run under. Keeping this in CFEngine policies could be a security hazard. Therefore, this option is not yet implemented on windows versions of CFEngine.

'exec_group'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The group name or id under which to run the process

Example:

```
body contain example
{
exec_group => "nogroup";
}
```

Notes:

This is part of the restriction of privilege for child processes when running cf-agent as the root group, or a group with privileges. It is ignored on windows, as processes do not have any groups associated with them.

'exec_timeout'

Type: int

Allowed input range: 1,3600

Synopsis: Timeout in seconds for command completion

Example:

```
body contain example
{
exec_timeout => "30";
}
```

Notes:

Attempt to time-out after this number of seconds. This cannot be guaranteed as not all commands are willing to be interrupted in case of failure.

'chdir' **Type**: string

Allowed input range: `"?(/.*)`

Synopsis: Directory for setting current/base directory for the process

Example:

```
body contain example

{
chdir => "/containment/directory";
}
```

Notes:

This command has the effect of placing the running command into a current working directory equal to the parameter given, i.e. it works like the 'cd' shell command.

'chroot' **Type**: string

Allowed input range: `"?(/.*)`

Synopsis: Directory of root sandbox for process

Example:

```
body contain example

{
chroot => "/private/path";
}
```

Notes:

Sets the path of the directory that will be experienced as the top-most root directory for the process. In security parlance, this creates a 'sandbox' for the process. Windows does not support this feature.

'preview' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false preview command when running in dry-run mode (with -n)

Default value: false

Example:

```
body contain example
{
preview => "true";
}
```

Notes:

Previewing shell scripts during a dry-run is a potentially misleading activity. It should only be used on scripts that make no changes to the system. It is CFEngine best practice to never write change-functionality into user-written scripts except as a last resort: CFEngine can apply its safety checks to user defined scripts.

'no_output'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false discard all output from the command

Default value: false

Example:

```
body contain example
{
no_output => "true";
}
```

Notes:

This is equivalent to piping standard output and error to '/dev/null'.

7.1.3 module

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false whether to expect the cfengine module protocol

Example:

```
commands:

   "/masterfiles/user_script"

     module => "true";
```

Notes:

If true, the module protocol is supported for this script, i.e. it is treated as a user module. A plug-in module may be written in any language, it can return any output you like, but lines which begin with a '+' sign are treated as classes to be defined (like '-D'), while lines which begin with a '-' sign are treated as classes to be undefined (like '-N'). Lines starting with '=' are scalar variables to be defined, and lines

beginning with '@' are lists. Any other lines of output are cited by cf-agent as being erroneous, so you should normally make your module completely silent. Here is an example written in shell:

```
#!/bin/sh
/bin/echo "@mylist= { \"one\", \"two\", \"three\" }"
/bin/echo "=myscalar= scalar val"
/bin/echo "+module_class"
```

And here is an example using it:

```
body common control
   {
   any::

      bundlesequence  => {
                         def,
                         modtest
                         };
   }

###################################################################

bundle agent def

{
commands:

  "$(sys.workdir)/modules/module_name" module => "true";

reports:

  #
  # Each module forms a private context with its name as id
  #

 module_class::

  "Module set variable $(module_name.myscalar)";

}

###################################################################

bundle agent modtest

{
vars:

 "mylist" slist => { @(module_name.mylist) };
```

```
reports:

 module_class::

  "Module set variable $(mylist)";

}
```

Here is an example module written in perl.

```
#!/usr/bin/perl
#
# module:myplugin
#

  # lots of computation....

if (special-condition)
   {
   print "+specialclass";
   }
```

If your module is "simple" and is best expressed as a shell command, then we suggest that you *expose* the class being defined in the command being executed (making it easier to see what classes are used when reading the promises file). For example, the promises could read as follows (the two `echo` commands are to ensure that the shell always exits with a successful execution of a command):

```
bundle agent sendmail
{
commands:
    # This next module checks a specific failure mode of dcc, namely
    # more than 3 error states since the last time we ran cf-agent
    is_mailhost::
        "/bin/test ‘/usr/bin/tail -100 /var/log/maillog | /usr/bin/grep ’Milter (dcc): to error stat
’’"
            contain => shell_command,
            module => "true";

    start_dccm::
        "/var/dcc/libexec/start-dccm"
            contain => not_paranoid;
}

body contain shell_command
{
    useshell    => "yes";
}

body contain not_paranoid
{
    useshell    => "no";
```

```
    exec_owner  => "root";
    umask       => "22";
}
```

Modules inherit the environment variables from cfagent and accept arguments, just as a regular command does.

```
#!/bin/sh
#
# module:myplugin
#

/bin/echo $*
```

Modules define variables in `cf-agent` by outputting strings of the form

```
=variablename=value
```

These variables end up in a context which has the same name as the module. When the `$(allclasses)` variable becomes too large to manipulate conveniently, you can access the complete list of currently defined classes in the file '`/var/cfengine/state/allclasses`'.

7.2 `databases` promises in '`agent`'

CFEngine Nova can interact with commonly used database servers to keep promises about the structure and content of data within them.

There are two main cases of database management to address: small embedded databases and large centralized databases.

CFEngine is a tool whose strength lies distributed management of computers. Databases are often centralized entities that have single point of management, so a large monolithic database is more easily managed with other tools. However, CFEngine can still monitor changes and discrepancies, and it can manage smaller embedded databases that are distributed in nature, whether they are SQL, registry or future types.

So creating 100 new databases for test purposes is a task for CFEngine, but adding a new item to an important production database is not a task that we recommend using CFEngine for.

There are three kinds of database supported by Nova:

LDAP - The Lightweight Directory Access Protocol
: A hierarchical network database primarily for reading simple schema.

SQL - Structured Query Language
: A number of relational databases (currently supported: MySQL, Postgres) for reading and writing complex data.

Registry - Microsoft Registry
: An embedded database for interfacing with system values in Microsoft Windows (Only CFEngine Nova)

In addition, CFEngine uses a variety of embedded databases for its own internals.

CFEngine's ability to make promises about databases depends on the good grace of the database server. Embedded databases are directly part of the system and promises can be made directly. However, databases running through a server process (either on the same host or on a different host) are independent agents and CFEngine cannot make promises on their behalf, unless they promise (grant) permission for CFEngine to make the changes. Thus the pre-requisite for making SQL database promises is to grant a point of access on the server.

```
databases:

 "database/subkey or table"

   database_operation => "create/delete/drop",
   database_type => "sql/ms_registry",
   database_columns => {
                       "name,type,size",
                       "name,type",
                       },

   database_server => body;


body database_server name
 {
 db_server_owner => "account name";
 db_server_password => "password";
 db_server_host => "hostname or omit for localhost";
 db_server_type => "mysql/posgres";
 db_server_connection_db => "database we can connect to";
 }
```

```
body common control
{
bundlesequence => { "databases" };
}

bundle agent databases

{
#commands:

#  "/usr/bin/createdb cf_topic_maps",

#        contain => as_user("mysql");

databases:

  "cf_topic_maps/topics"
```

```
    database_operation => "create",
    database_type => "sql",
    database_columns => {
                        "topic_name,varchar,256",
                        "topic_comment,varchar,1024",
                        "topic_id,varchar,256",
                        "topic_type,varchar,256",
                        "topic_extra,varchar,26"
                        },

    database_server => myserver;



}

################################################

body database_server myserver
{
any::
 db_server_owner => "postgres";
 db_server_password => "";
 db_server_host => "localhost";
 db_server_type => "postgres";
 db_server_connection_db => "postgres";
none::
 db_server_owner => "root";
 db_server_password => "";
 db_server_host => "localhost";
 db_server_type => "mysql";
 db_server_connection_db => "mysql";
}

body contain as_user(x)
{
exec_owner => "$(x)";
}
```

The promiser in database promises is a concatenation of the database name and underlying tables. This presents a simple hierarchical model that looks like a file-system. This is the normal structure within the Windows registry for instance. Entity-Relation databases do not normally present tables in this way, but no harm is done in representing them as a hierarchy of depth 1.

7.2.1 database_server (body template)

Type: (ext body)

'db_server_owner'

Type: string

Allowed input range: (arbitrary string)

Synopsis: User name for database connection

Example:

```
db_server_owner => "mark";
```

Notes:

'db_server_password'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Clear text password for database connection

Example:

```
db_server_password => "xyz.1234";
```

Notes:

'db_server_host'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Hostname or address for connection to database, blank means localhost

Example:

```
db_server_host => "sqlserv.example.org";
```

Notes:

Hostname or IP address of the server.

'db_server_type'

Type: (menu option)

Allowed input range:

```
postgres
mysql
```

Synopsis: The dialect of the database server

Default value: none

Example:

```
db_server_type => "postgres";
```

Notes:

'db_server_connection_db'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The name of an existing database to connect to in order to create/manage other databases

Example:

```
body database_server myserver(x)
{
db_server_owner => "$(x)";
db_server_password => "";
db_server_host => "localhost";
db_server_type => "$(mysql)";
db_server_connection_db => "$(x)";
}
```

where 'x' is currently mysql or postgres.

Notes:

In order to create a database on a database server (all of which practice voluntary cooperation), one has to be able to connect to the server, however, without an existing database this is not allowed. Thus, database servers provide a default database that can be connected to in order to thereafter create new databases. These are called postgres and mysql for their respective database servers.

For the knowledge agent, this setting is made in the control body, for database verification promises, it is made in the `database_server` body.

7.2.2 database_type

Type: (menu option)
Allowed input range:

```
sql
ms_registry
```

Default value: none
Synopsis: The type of database that is to be manipulated
Example:

```
database_type => "ms_registry";
```

Notes:

7.2.3 database_operation

Type: (menu option)
Allowed input range:

```
create
delete
drop
cache
verify
restore
```

Synopsis: The nature of the promise - to be or not to be
Example:

```
database_operation => "create";
```

Notes:

7.2.4 database_columns

Type: slist
Allowed input range: `.*`

Synopsis: A list of column definitions to be promised by SQL databases

Example:

```
  "cf_topic_maps/topics"

    database_operation => "create",
    database_type => "sql",
    database_columns => {
                        "topic_name,varchar,256",
                        "topic_comment,varchar,1024",
                        "topic_id,varchar,256",
                        "topic_type,varchar,256",
                        "topic_extra,varchar,26"
                        },

    database_server => myserver;
```

Notes:

Columns are a list of tuplets (*Name,type,size*). Array items are triplets, and fixed size data elements are doublets.

7.2.5 database_rows

Type: slist

Allowed input range: `.*,.*`

Synopsis: An ordered list of row values to be promised by SQL databases

Example:

```
bundle agent databases

{
databases:

 windows::

  # Regsitry has (value,data) pairs in "keys" which are directories

  "HKEY_LOCAL_MACHINE\SOFTWARE\CFEngine AS\CFEngine"

    database_operation => "create",
    database_rows => { "value1,REG_SZ,new value 1", "value2,REG_DWORD,12345"} ,
```

```
    database_type     => "ms_registry";
}
```

Notes:

This constraint is used only in adding data to database columns. Rows are considered to be instances of individual columns.

In the case of the system registry on Windows, the rows represent data on data-value pairs. The currently supported types (the middle field) for the Windows registry are `REG_SZ` (string), `REG_EXPAND_SZ` (expandable string) and `REG_DWORD` (double word).

7.2.6 `registry_exclude`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of regular expressions to ignore in key/value verification

Example:

```
databases:

 "HKEY_LOCAL_MACHINE\SOFTWARE"

    database_operation => "cache",

    registry_exclude => { ".*Windows.*CurrentVersion.*",
                          ".*Touchpad.*",
                          ".*Capabilities.FileAssociations.*",
                          ".*Rfc1766.*" ,
                          ".*Synaptics.SynTP.*",
                          ".*SupportedDevices.*8086",
                          ".*Microsoft.*ErrorThresholds"
                        },

    database_type     => "ms_registry";
```

Notes:

During recursive Windows registry scanning, this option allows us to ignore keys of values matching a list of regular expressions. Some values in the registry are ephemeral and some should not be considered. This provdes a convenient way of avoiding names. It is analogous to `exclude_dirs` for files.

7.3 guest_environments promises in 'agent'

Guest environment promises describe enclosed computing environments that can host physical and virtual machines, solaris zones, grids, clouds or other enclosures, including embedded systems. CFEngine will support the convergent maintenance of such inner environments in a fixed location, with interfaces to an external environment.

CFEngine currently seeks to add convergence properties to existing interfaces for automatic self-healing of guest environments. The current implementation integrates with *libvirt*, supporting host virtualization for Xen, KVM, VMWare, etc. Thus CFEngine, running on a virtual host, can maintain the state and deployment of virtual guest machines defined within the *libvirt* framework. Guest environment promises are not meant to manage what goes on within the virtual guests: for that purpose, you should run CFEngine directly on the virtual machine, as if it were any other machine.

```
site1::

 "unique_name1"

      environment_resources => myresources("2GB","512MB"),
      environment_interface => mymachine("hostname"),
           environment_type => "xen",
           environment_state => "running",
           environment_host => "atlas";

 "unique_name2"

           environment_type => "xen_network",
          environment_state => "create",
           environment_host => "atlas";
```

CFEngine currently provides a convergent interface to *libvirt*.

7.3.1 environment_host

Type: string

Allowed input range: [a-zA-Z0-9_]+

Synopsis: A class indicating which physical node will execute this guest machine

Example:

```
guest_environments:
```

```
linux::

"host1"
                comment => "Keep this vm suspended",
  environment_resources => myresources,
       environment_type => "kvm",
      environment_state => "suspended",
       environment_host => "ubuntu";
```

Notes:

The promise will only apply to the machine with this class set. Thus, CFEngine must be running locally on the hypervisor for the promise to take effect.

This attribute is required.

History: this feature was introduced in Nova 2.0.0 (2010), Community 3.3.0 (2012)

7.3.2 `environment_interface` (body template)

Type: (ext body)

'`env_addresses`'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: The IP addresses of the environment's network interfaces

Example:

```
body environment_interface vnet(primary)
{
env_name      => "$(this.promiser)";
env_addresses => { "$(primary)" };

host1::

  env_network => "default_vnet1";

host2::

  env_network => "default_vnet2";

}
```

Notes:

The IP addresses of the virtual machine can be overridden here at run time.

'env_name' **Type**: string

Allowed input range: (arbitrary string)

Synopsis: The hostname of the virtual environment

Example:

```
body environment_interface vnet(primary)
{
env_name      => "$(this.promiser)";
env_addresses => { "$(primary)" };

host1::
  env_network => "default_vnet1";

host2::
  env_network => "default_vnet2";
}
```

Notes:

The 'hostname' of a virtual guest may or may not be the same as the identifier used as 'promiser' by the virtualization manager.

'env_network'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The hostname of the virtual network

Example:

```
body environment_interface vnet(primary)
     {
     env_name      => "$(this.promiser)";
     env_addresses => { "$(primary)" };

     host1::
       env_network => "default_vnet1";

     host2::
       env_network => "default_vnet2";
     }
```

Notes:

7.3.3 `environment_resources` (body template)

Type: (ext body)

'env_cpus' **Type**: int

Allowed input range: `0,99999999999`

Synopsis: Number of virtual CPUs in the environment

Example:

```
body environment_resources my_environment
{
env_cpus => "2";
env_memory => "512"; # in KB
env_disk => "1024";  # in MB
}
```

Notes:

The maximum number of cores or processors in the physical environment will set a natural limit on this value.

This attribute conflicts with `env_spec`.

'env_memory'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Amount of primary storage (RAM) in the virtual environment (KB)

Example:

```
body environment_resources my_environment
{
env_cpus => "2";
env_memory => "512"; # in KB
env_disk => "1024";  # in MB
}
```

Notes:

The maximum amount of memory in the physical environment will set a natural limit on this value.

This attribute conflicts with env_spec.

'env_disk'

Type: int

Allowed input range: 0,99999999999

Synopsis: Amount of secondary storage (DISK) in the virtual environment (MB)

Example:

```
body environment_resources my_environment
{
env_cpus => "2";
env_memory => "512"; # in KB
env_disk => "1024";  # in MB
}
```

Notes:

This parameter is currently unsupported, for future extension.

This attribute conflicts with env_spec.

'env_baseline'

Type: string

Allowed input range: "?(/.*)

Synopsis: The path to an image with which to baseline the virtual environment

Example:

```
env_baseline => "/path/to/image";
```

Notes:

This function is for future development.

'env_spec'

Type: string

Allowed input range: .*

Synopsis: A string containing a technology specific set of promises for the virtual instance

Example:

```
body environment_resources virt_xml(host)
{
env_spec =>

"<domain type='xen'>
  <name>$(host)</name>
  <os>
    <type>linux</type>
    <kernel>/var/lib/xen/install/vmlinuz-ubuntu10.4-x86_64</kernel>
    <initrd>/var/lib/xen/install/initrd-vmlinuz-ubuntu10.4-x86_64</initrd>
    <cmdline> kickstart=http://example.com/myguest.ks </cmdline>
  </os>
  <memory>131072</memory>
  <vcpu>1</vcpu>
  <devices>
    <disk type='file'>
      <source file='/var/lib/xen/images/$(host).img'/>
      <target dev='sda1'/>
    </disk>
    <interface type='bridge'>
      <source bridge='xenbr0'/>
      <mac address='aa:00:00:00:00:11'/>
      <script path='/etc/xen/scripts/vif-bridge'/>
    </interface>
    <graphics type='vnc' port='-1'/>
    <console tty='/dev/pts/5'/>
  </devices>
</domain>
";
}
```

Notes:

The preferred way to specify the resources of an environment on creation, i.e. when `envrionment_state` is '`create`'.

This attibute conflicts with `env_cpus`, `env_memory` and `env_disk`.

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

7.3.4 `environment_state`

Type: (menu option)

Allowed input range:

```
create
delete
running
suspended
```

```
down
```

Synopsis: The desired dynamical state of the specified environment

Example:

```
guest_environments:

 linux::

 "bishwa-kvm1"
                  comment => "Keep this vm suspended",
    environment_resources => myresources,
         environment_type => "kvm",
        environment_state => "suspended",
         environment_host => "ubuntu";
```

Notes:

The allowed states have the following convergent semantics.

'create' The guest machine is allocated, installed and left in a running state.

'delete' The guest machine is shut down and de-allocated but no files are removed.

'running' The guest machine is in a running state, if it previously exists.

'suspended'
The guest exists in a suspended state or a shutdown state. If the guest is running, it is suspended, else it is ignored.

'down' The guest machine is shut down, but not de-allocated.

7.3.5 environment_type

Type: (menu option)

Allowed input range:

```
xen
kvm
esx
vbox
test
xen_net
kvm_net
esx_net
test_net
zone
```

```
                ec2
                eucalyptus
```

Synopsis: Virtual environment type

Example:

```
bundle agent my_vm_cloud
{
guest_environments:

 scope::

   "vguest1"

       environment_resources => my_environment_template,
       environment_interface => vnet("eth0,192.168.1.100/24"),
       environment_type      => "test",
       environment_state     => "create",
       environment_host      => "atlas";

   "vguest2"

       environment_resources => my_environment_template,
       environment_interface => vnet("eth0,192.168.1.101/24"),
       environment_type      => "test",
       environment_state     => "delete",
       environment_host      => "atlas";
}
```

Notes:

The currently supported types are those supported by *libvirt*. More will be added as time goes on.

7.4 `files` promises in '`agent`'

Files promises are an umbrella concept for all attributes of files. Operations fall basically into three categories: create, delete and edit.

```
files:

  "/path/file_object"

      perms => perms_body,
      ... ;
```

Prior to version 3, file promises were scattered into many different types such as `files`, `tidy`, `copy`, `links`, etc. File handling in CFEngine 3 uses regular expressions everywhere for pattern matching. The old 'wildcard/globbing' expressions '`*`' and '`?`' are deprecated, and everything is based consistently on Perl Compatible Regular Expressions.

There is a natural ordering in file processing that obviates the need for the actionsequence. The trick of using multiple actionsequence items with different classes, e.g.

```
actionsequence = ( ... files.one  ..  files.two )
```

can now be handled more elegantly using bundles. The natural ordering uses that fact that some operations are mutually exclusive and that some operations do not make sense in reverse order. For example, editing a file and then copying onto it would be nonsense. Similarly, you cannot both remove a file and rename it.

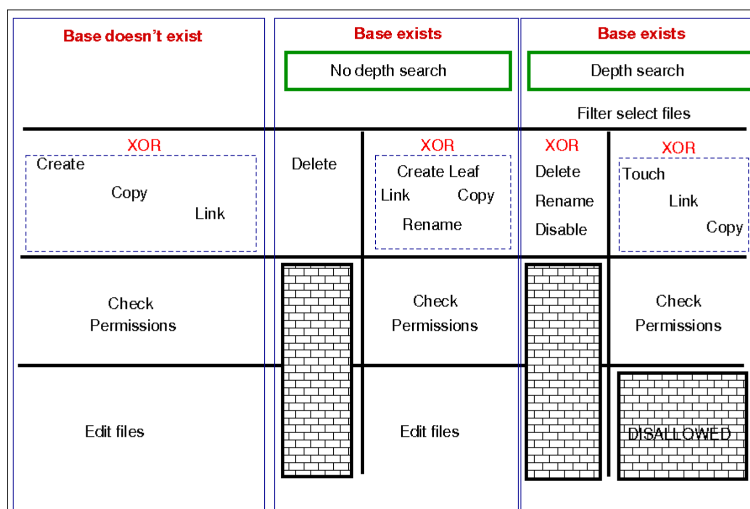
File copying

One of the first things users of CFEngine 2 will notice is that copying is now 'backwards'. Instead of the default object being source and the option being the destination, in CFEngine 3 the destination is paramount and the source is an option. This is because the model of voluntary cooperation tells us that it is the object that is changed which is the agent making the promise. One cannot force change onto a destination with CFEngine, one can only invite change from a source.

Normal ordering of promise attributes

CFEngine 3 no longer has an 'action sequence'. Ordering of operations has, in most cases, a natural ordering which is assumed by the agent. For instance: 'delete then create' (normal ordering) makes sense, but 'create then delete' does not. This sort of principle can be extended to deal with all aspects of file promises.

The diagram below shows the ordering. Notice that the same ordering applies regardless of file type (plain-file or directory). Note also that file editing is done "atomically" (see 'File editing in CFEngine 3' for important details).



The pseudo-code for this logic is shown in the diagram and below:

```
for each file promise-object
   {
   if (depth_search)
     do
```

```
        DepthSearch (HandleLeaf)
      else
        (HandleLeaf)
      done
    }

 HandleLeaf()
   {
   Does leaf-file exist?

     NO:  create
     YES: rename,delete,touch,

     do
      for all servers in {localhost, @(servers)}
         {
         if (server-will-provide)
            do
              if (depth_search)
                 embedded source-depth-search (use file source)
                 break
              else
                 (use file source)
                 break
              done
            done
         }
     done

   Do all links (always local)

   Check Permissions

   Do edits
   }
```

Depth searches (recursion) during searches

In CFEngine 2 there was the concept of recursion during file searches. Recursion is now called "depth-search". In addition, it was possible to specify wildcards in the base-path for this search. CFEngine 3 replaces the 'globbing' symbols with standard regular expressions:

```
      CFEngine 2               CFEngine 3

/one/*/two/thr*/four    /one/.*/two/thr.*/four
```

Note that this now means when searching for "hidden" files (files with names starting with a '.') or files with specific extensions, you should take care to escape the dot (e.g., `\.cshrc` or `.*\.txt` when you wish it to mean a literal character and not the "any character" interpretation provided by regular expression interpretation. Note that when you do a recursive search, the files '`'.'`' and '`'..'`' are never included in the matched files, even if the regular expresion in the '`leaf_name`' specifically allows them.

Note also the the filename '`/dir/ect/ory/.`' is a special case used with the '`create`' attribute to indicate the directory named '`/dir/ect/ory`' and not any of the files under it. If you really want to specify a regular expression that matches any single-character filename, use '`/dir/ect/ory/[\w\W]`' as your promise regular expression (you can't use '`/dir/ect/ory[^/]`', see below for an explanation.

When we talk about a depth search, it refers to a search for file objects which starts from the one or more matched base-paths as shown in the example above.

Filenames and regular expressions

CFEngine allows regular expressions within filenames, but only after first doing some sanity checking to prevent some readily avoidable problems. The biggest rule you need to know about filenames and regular expressions is that *all* regular expressions in filenames are bounded by directory separators, and that each component expression is anchored between the directory separators See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39. In other words, CFEngine splits up any file paths into its component parts, and then it evaluates any regular expressions at a component-level.

What this means is that the path '`/tmp/gar.*`' will only match filenames like '`/tmp/gar`', '`/tmp/garbage`' and '`/tmp/garden`'. It will *not* match filename like '`/tmp/gar/baz`' (because even though the '`.*`' in a regular expression means "zero or more of any character", CFEngine restricts that to mean "zero or more of any character *in a path component*"). Correspondingly, CFEngine also restricts where you can use the '`/`' character (you can't use it in a character class like '`[^/]`' or in a parenthesized or repeated regular expression component.

This means that regular expressions which include "optional directory components" won't work. You can't have a files promise to tidy the directory '`(/usr)?/tmp`'. Instead, you need to be more verbose and specify '`/usr/tmp|/tmp`', or even better, think declaratively and create an *slist* that contains both the strings '`/tmp`' and '`/usr/tmp`', and then allow CFEngine to iterate over the list!

This also means that the path '`/tmp/.*/something`' will match files like '`/tmp/abc/something`' or '`/tmp/xyzzy/something`'. However, even though the pattern '`.*`' means "zero or more of any character (except '/')", CFEngine matches files bounded by directory separators. So even though the pathname '`/tmp//something`' is technically the same as the pathname '`/tmp/something`', the regular expression '`/tmp/.*/something`' will *not* match on the degenerate case of '`/tmp//something`' (or '`/tmp/something`').

Promises involving regular expressions

CFEngine can only keep (or repair, or fail to keep) a promise on files which actually exist. If you make a promise based on a wildcard match, then the promise is only ever attempted if the match succeeds. However, if you make a promise which includes a recursive search which includes a wildcard match, then the promise can be kept or repaired (provided that the directory specified in the promise exists). Consider the following two examples, (assuming that there first exist files named '`/tmp/gar`', '`/tmp/garbage`' and '`/tmp/garden`'). At first blush, the two promises look like they should do the same thing, but there is a subtle difference:

```
bundle agent foobaz                    bundle agent foobaz
{                                      {
files:                                 files:
```

```
 "/tmp/gar.*"
    delete => tidy,
    classes => if_ok("done");
```

```
 "/tmp"
     delete => tidy,
     depth_search => recurse("0"),
     file_select => gars,
     classes => if_ok("done");
}

body file_select gars
{
     leaf_name => { "gar.*" };
     file_result => "leaf_name";
}
```

```
body classes if_ok(x)
{
promise_repaired => { "$(x)" };
promise_kept => { "$(x)" };
}
```

```
body classes if_ok(x)
{
promise_repaired => { "$(x)" };
promise_kept => { "$(x)" };
}
```

In the first example, when the configuration containing this promise is first executed, any file starting with "gar" that exists in the '/tmp' directory will be removed, and the 'done' class will be set. However, when the configuration is executed a second time, the pattern '/tmp/gar.*' will not match any files, and that promise will not even be *attempted* (and, consequently the 'done' class will *not* be set).

In the second example, when the configuration containing this promise is first executed, any file starting with "gar" that exists in the '/tmp' directory will also be removed, and the 'done' class will also be set. The second time the configuration is executed, however, the promise on the '/tmp' directory will still be executed (because '/tmp' of course still exists), and the 'done' class *will* be set, because all files matching the 'file_select' attribute have been deleted from that directory.

Local and remote searches

There are two distinct kinds of depth search:

- A local search over promiser agents.
- A remote search over provider agents.

When we are *copying* or *linking* to a file source, it is the search over the *remote* source that drives the content of a promise (the promise is a promise to use what the remote source provides). In general, the sources are on a different device to the images that make the promises. For all other promises, we search over existing local objects.

If we specify depth search together with copy of a directory, then the implied remote source search is assumed, and it is made after the search over local base-path objects has been made. If you mix complex promise body operations in a single prmose, this could lead to confusion about the resulting behaviour, and a warning is issued. In general it is not recommended to mix searches without a full understanding of the consequences, but this might occasionally be useful.

Depth search is not allowed with edit_line promises.

File editing in CFEngine 3

CFEngine 2 assumed that all files were line-edited, because it was based on Unix traditions. Since then many new file formats have emerged, including XML. CFEngine 3 opens up the possibiltiy for

multiple models of file editing. Line based editing is still present and is both much simplified and much more powerful than previously.

File editing is not just a single kind of promise but a whole range of 'promises within files'. It is therefore not merely a body to a single kind of promise, but a bundle of sub-promises. After all, inside each file is a new world of objects that can make promises, quite separate from files' external attributes.

A typical file editing stanza has the elements in the following example.

```
########################################################################
#
# File editing
#
########################################################################

body common control

{
version => "1.2.3";
bundlesequence  => { "outerbundle"  };
}

########################################################

bundle agent outerbundle

{
files:

  "/home/mark/tmp/cf3_test"

       create    => "true",     # Like autocreate in cf2
       edit_line => inner_bundle;
}

########################################################

bundle edit_line inner_bundle
  {
  vars:

   "who" string => "SysAdmin John";# private variable in bundle

  insert_lines:
    "/* This file is maintained by CFEngine (see $(who) for details) */",
         location => first_line;

  replace_patterns:
```

```
   # replace shell comments with C comments

   "#(.*)"

      replace_with => C_comment,
     select_region => MySection("New section");

  reports:

    someclass::

      "This is file $(edit.filename)";
  }

########################################
# Bodies for the library ...
########################################

body replace_with C_comment

{
replace_value => "/* $(match.1) */"; # backreference
occurrences => "all";          # first, last all
}

########################################################

body select_region MySection(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}

########################################################

body location first_line

{
before_after => "before";
first_last => "first";
select_line_matching => ".*";
}
```

There are several things to notice:

- The line-editing promises are all convergent promises about patterns within the file. They have bodies, just like other attributes do and these allow us to make simple templates about file editing while extending the power of the basic primitives.
- All file edits specified in a single `edit_line` bundle are handled "atomically". CFEngine edits files like this:
  - CFEngine makes a copy of the file you you want to edit
  - CFEngine makes all the edits in the **copy** of the file. The filename is the same as your original file with the extension '`.cf-after-edit`' appended.
  - After all edits are complete (the `delete_lines`, `field_edits`, `insert_lines`, and finally `replace_patterns` promises), CFEngine checks to see if the new file is the same as the original one. If there are no differences, the promises have converged, so it deletes the copy, and the original is left completely unmodified.
  - If there are any differences, CFEngine makes a copy of your original file with the extension '`.cf-before-edit`' (so you always have the most recent backup available), and then renames the edited version to your original filename.

  Because file rename is an atomic operation (guaranteed by the operating system), any application program will either see the old version of the file or the new one – there is no "window of opportunity" where a partially edited file can be seen (unless an application intentionally looks for the '`.cf-after-edit`' file). Problems during editing (such as disk-full or permission errors) are likewise detected, and CFEngine will not rename a partial file over your original.
- All pattern matching is through perl compatible regular expressions
- Editing takes place within a marked region (which defaults to the whole file if not otherwise specified).
- Search/replace functions now allow back-references.
- The line edit model now contains a field or column model for dealing with tabular files such as Unix '`passwd`' and '`group`' files. We can now apply powerful convergent editing operations to single fields inside a table, to append, order and delete items from lists inside fields.
- The special variable `$(edit.filename)` contains the name of the file being edited within an edit bundle.

In the example above, back references are used to allow conversion of comments from shell-style to C-style.

Another example of files promises is to look for changes in files. The following example reports on all recent changes to files in a directory by maintaining the most recent version of the `md5` hash of the file contents. Similar checks can be used to examine metadata or both the contents and metadata, as well as using different difference checks. The Community Edition only reports that changes were found, but Enterprise versions of CFEngine can also report on what exactly the significant changes were.

```
bundle agent example
{
files:

  "/home/mark/tmp" -> "Security team"
```

```
       changes      => lay_a_tripwire,
       depth_search => recurse("inf"),
       action       => background;
}

#########################################################

body changes lay_a_tripwire

{
hash           => "md5";
report_changes => "content";
update         => "yes";
}
```

7.4.1 `acl` (body template)

Type: (ext body)

'`aces`'

Type: slist

Allowed input range: `((user|group):[^:]+:[-=+,rwx()dtTabBpcoD]*(:(allow|deny))?)|((all|mask)` `=+,rwx()]*(:(allow|deny))?)`

Synopsis: Native settings for access control entry

Example:

```
body acl template

{
acl_method => "overwrite";
acl_type => "posix";
acl_directory_inherit => "parent";

aces => {
        "user:*:r(wwx),-r:allow",
        "group:*:+rw:allow",
        "mask:x:allow",
        "all:r"
        };
}
```

Notes:

POSIX ACL are available in CFEngine Community starting with 3.4.0. NTFS ACL are available with CFEngine Nova or above. Form of the permissions is:

```
aces => {
        "user:uid:mode[:perm_type]", ...,
        "group:gid:mode[:perm_type]", ...,
        "all:mode[:perm_type]"
        };
```

- `user` indicates that the line applies to a user specified by the user identitfier `uid`. `mode` is the permission mode string.
- `group` indicates that the line applies to a group specified by the group identitfier `gid`. `mode` is the permission mode string.
- `all` indicates that the line applies to every user. `mode` is the permission mode string.
- `uid` is a valid user identifier for the system and cannot be empty. However, `uid` can be set to * as a synonym for the entity that owns the file system object (e.g. user:*:r).
- `gid` is a valid group identifier for the system and cannot be empty. However, in some acl types, `gid` can be set to * to indicate a special group (e.g. in POSIX this refers to the file group).
- `mode` is one or more strings `op|perms|(nperms)`; a concatenation of `op`, `perms` and optionally (`nperms`), see below, separated with commas (e.g. +rx,-w(s)). `mode` is parsed from left to right.
- `op` specifies the operation on any existing permissions, if the defined ACE already exists. `op` can be =, empty, + or -. = or empty sets the permissions to the ACE as stated, + adds and - removes the permissions from any existing ACE.
- `nperms` (optional) specifies file system specific (native) permissions. Only valid if `acl_type` is defined. `nperms` will only be enforced if the file object is stored on a file system supporting the acl type set in `acl_type` (e.g. `nperms` will be ignored if `acl_type:ntfs` and the object is stored on a file system not supporting ntfs ACLs). Valid values for `nperms` varies with different ACL types, and is defined in subsequent sections.
- `perm_type` (optional) can be set to either `allow` or `deny`, and defaults to `allow`. `deny` is only valid if `acl_type` is set to an ACL type that support deny permissions. A `deny` ACE will only be enforced if the file object is stored on a file system supporting the acl type set in `acl_type`.

`gperms` (generic permissions) is a concatenation of zero or more of the characters shown in the table below. If left empty, none of the permissions are set.

Flag	Description	Semantics on file	Semantics on directory

r	Read	Read data, permissions, attributes	Read directory contents, permissions, attributes
w	Write	Write data	Create, delete, rename subobjects
x	Execute	Execute file	Access subobjects

Note that the r permission is not neccessary to read an object's permissions and attributes in all file systems (e.g. in POSIX, having x on its containing directory is sufficient).

'acl_directory_inherit'

Type: (menu option)

Allowed input range:

```
nochange
parent
specify
clear
```

Synopsis: Access control list type for the affected file system

Example:

```
body acl template

{
acl_method => "overwrite";
acl_type => "posix";
acl_directory_inherit => "parent";

aces => {
        "user:*:rwx:allow",
        "group:*:+rw:allow",
        "mask:rx:allow",
        "all:r"
        };
}
```

Notes:

Directories have ACLs associated with them, but they also have the ability to inherit an ACL to sub-objects created within them. POSIX calls the former ACL type "access ACL" and the latter "default ACL", and we will use the same terminology.

The constraint acl_directory_inherit gives control over the default ACL of directories. The default ACL can be left unchanged (nochange), empty (clear), or be explicitly specified (specify). In addition, the default ACL can be set equal to the directory's

access ACL (parent). This has the effect that child objects of the directory gets the same access ACL as the directory.

'acl_method'

Type: (menu option)

Allowed input range:

```
append
overwrite
```

Synopsis: Editing method for access control list

Example:

```
body acl template

{
acl_method => "overwrite";
acl_type => "posix";
aces => { "user:*:rw:allow", "group:*:+r:allow", "all:"};
}
```

Notes:

When defining an ACL, we can either use an existing ACL as the starting point, or state all entries of the ACL. If we just care about one entry, say that the superuser has full access, the method constraint can be set to append, which is the default. This has the effect that all the existing ACL entries that are not mentioned will be left unchanged. On the other hand, if method is set to overwrite, the resulting ACL will only contain the mentioned entries. When doing this, it is important to check that all the required ACL entries are set, e.g. owning user, group and all in Posix ACLs.

'acl_type' **Type**: (menu option)

Allowed input range:

```
generic
posix
ntfs
```

Synopsis: Access control list type for the affected file system

Example:

```
body acl template
```

```
{
acl_type => "ntfs";
aces => { "user:Administrator:rwx(po)", "user:Auditor:r(o)"};
}
```

Notes:

ACLs are supported on multiple platforms, which may have different sets of available permission flags. By using the constraint `acl_type`, we can specify which platform, or ACL API, we are targeting with the ACL. The default, `generic`, is designed to work on all supported platforms. However, if very specific permission flags are required, like "Take Ownership" on the NTFS platform, we must set `acl_type` to indicate the target platform. Currently, the supported values are `posix` and `ntfs`.

'`specify_inherit_aces`'

Type: slist

Allowed input range: `((user|group):[^:]+:[-=+,rwx()dtTabBpcoD]*(:(allow|deny))?)|((all|mask)` `=+,rwx()]*(:(allow|deny))?)`

Synopsis: Native settings for access control entry

Example:

```
body acl template
{
specify_inherit_aces => {  "all:r" };
}
```

Notes:

`specify_inherit_aces` (optional) is a list of access control entries that are set on child objects. It is also parsed from left to right and allows multiple entries with same entity-type and id. Only valid if `acl_directory_inherit` is set to `specify`.

This is an acl which makes explicit setting for the acl inherited by new objects within a directory. It is included for those implementations that do not have a clear inheritance policy.

7.4.2 `changes` (body template)

Type: (ext body)

'`hash`' **Type**: (menu option)

Allowed input range:

`md5`

```
sha1
sha224
sha256
sha384
sha512
best
```

Synopsis: Hash files for change detection

Example:

```
body changes example
{
hash => "md5";
}
```

Notes:

The best option cross correlates the best two available algorithms known in the OpenSSL library.

'report_changes'

Type: (menu option)

Allowed input range:

```
all
stats
content
none
```

Synopsis: Specify criteria for change warnings

Example:

```
body changes example
{
report_changes => "content";
}
```

Notes:

Files can change in permissions and contents, i.e. external or internal attributes. If 'all' is chosen all attributes are checked.

'update_hashes'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Update hash values immediately after change warning

Example:

```
body changes example
{
update_hashes => "true";
}
```

Notes:

If this is positive, file hashes should be updated as soon as a change is registered so that multiple warnings are not given about a single change. This applies to addition and removal too.

'report_diffs'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Generate reports summarizing the major differences between individual text files

Example:

```
body changes example
{
```

```
report_diffs => "true";
}
```

Notes:

This feature is available only in enterprise levels Nova and above.

If true, CFEngine will log a 'diff' summary of major changes to the files. It is not permitted to combine this promise with a depth search, since this would consume a dangerous amount of resources and would lead to unreadable reports.

The feature is intented as a informational summary, not as a version control function suitable for transaction control. If you want to do versioning on system files, you should keep a single repository for them and use CFEngine to synchronize changes from the repository source. Repositories should not be used to attempt to capture random changes of the system.

7.4.3 copy_from (body template)

Type: (ext body)

'source' **Type**: string

Allowed input range: `.+`

Synopsis: Reference source file from which to copy

Example:

```
body copy_from example
{
source => "/path/to/source";
}

# or

body link_from example
{
source => "/path/to/source";
}
```

Notes:

For remote copies this refers to the file name on the remote server.

'servers' **Type**: slist

Allowed input range: `[A-Za-z0-9_.:-]+`

Synopsis: List of servers in order of preference from which to copy

Example:

```
body copy_from example
{
servers => { "primary.example.org", "secondary.example.org",
                 "tertiary.other.domain" };
}
```

Notes:

The servers are tried in order until one of them succeeds.

'collapse_destination_dir'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false Place files in subdirectories into the root destination directory during copy

Example:

```
body copy_from mycopy(from,server)

{
source      => "$(from)";
servers     => { "$(server)" };
collapse_destination_dir => "true";
}
```

Notes:

Under normal operations, recursive copies cause CFEngine to track subdirectories of files. So, for instance, if we copy recurively from 'src' to 'dest', then 'src/subdir/file' will map to 'dest/subdir/file'.

By setting this option to 'true', the promiser destination directory promises to aggregate files searched from all subdirectories into itself, i.e. a single destination directory.

'compare' **Type**: (menu option)

Allowed input range:

```
atime
mtime
ctime
digest
hash
exists
binary
```

Synopsis: Menu option policy for comparing source and image file attributes

Default value: mtime or ctime differs

Example:

```
body copy_from example

{
compare => "digest";
}
```

Notes:

The default copy method is 'mtime' (modification time), meaning that the source file is copied to the destination (promiser) file, if the source file has been modified more recently than the destination.

The different options are:

- `mtime` CFEngine copies the file if the modification time of the source file is more recent than that of the promised file
- `ctime` CFEngine copies the file if the creation time of the source file is more recent than that of the promised file
- `atime` CFEngine copies the file if the modification time or creation time of the source file is more recent than that of the promised file. If the times are equal, a byte-for-bye comparison is done on the files to determine if it needs to be copied.
- `exists` CFEngine copies the file if the promised file does not already exist.
- `binary` CFEngine copies the file if they are both plain files and a byte-for-byte comparison determines that they are different. If both are not plain files, CFEngine reverts to comparing the `mtime` and `ctime` of the files. If the source file is on a different machine (i.e., network copy), then `hash` is used instead to reduce network bandwidth.

- `hash` CFEngine copies the file if they are both plain files and a message digest comparison indicates that the files are different. In Enterprise versions of CFEngine version 3.1.0 and later, SHA256 is used as a message digest hash to conform with FIPS; in older Enterprise versions of CFEngine and all Community versions, MD5 is used.
- `digest` a synonym for `hash`

'`copy_backup`'

Type: (menu option)

Allowed input range:

```
true
false
timestamp
```

Synopsis: Menu option policy for file backup/version control

Default value: true

Example:

```
body copy_from example
{
copy_backup => "timestamp";
}
```

Notes:

Determines whether a backup of the previous version is kept on the system. This should be viewed in connection with the system repository, since a defined repository affects the location at which the backup is stored. See Section 5.2.33 [default_repository], page 89 and Section 7.4.17 [repository], page 277 for further details.

'`encrypt`' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false use encrypted data stream to connect to remote host

Default value: false

Example:

```
body copy_from example
{
servers  => { "remote-host.example.org" };
encrypt => "true";
}
```

Notes:

Client connections are encrypted with using a Blowfish randomly generated session key. The intial connection is encrypted using the public/private keys for the client and server hosts.

'check_root'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false check permissions on the root directory when depth_search

Example:

```
body copy_from example
{
check_root => "true";
}
```

Notes:

When copying files recursively (by depth search), this flag determines whether the permissions of the root directory should be set from the root of the source. The default is to check only copied file objects and subdirectories within this root (false).

'copylink_patterns'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of patterns matching files that should be copied instead of linked

Example:

```
body copy_from example
{
copylink_patterns => { "special_node1", "other_node.*" };
}
```

Notes:

The matches are performed on the last node of the filename, i.e. the file without its path. As windows does not support symbolic links, this feature is not available there.

'copy_size'

Type: irange [int,int]

Allowed input range: `0,inf`

Synopsis: Integer range of file sizes that may be copied

Default value: any size range

Example:

```
body copy_from example
{
copy_size => irange("0","50000");
}
```

Notes:

The use of the irange function is optional. Ranges may also be specified as a comma separated numbers.

'findertype'

Type: (menu option)

Allowed input range:

```
MacOSX
```

Synopsis: Menu option for default finder type on MacOSX

Example:

```
body copy_from example
{
findertype => "MacOSX";
}
```

Notes:

This applies only to the Macintosh OSX variants.

'linkcopy_patterns'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of patterns matching files that should be replaced with symbolic links

Example:

```
body copy_from mycopy(from)

{
source            => "$(from)";
linkcopy_patterns => { ".*" };
}
```

Notes:

The pattern matches the last node filename (i.e. without the absolute path). Windows only supports hard links, see `link_type`.

'link_type'

Type: (menu option)

Allowed input range:

```
symlink
hardlink
relative
absolute
```

Synopsis: Menu option for type of links to use when copying

Default value: symlink

Example:

```
body link_from example
{
link_type => "symlink";
source => "/tmp/source";
}
```

Notes:

What kind of link should be used to link files. Users are advised to be wary of 'hard links' (see Unix manual pages for the 'ln' command). The behaviour of non-symbolic links is often precarious and unpredictable. However, hard links are the only supported type by windows.

Note that 'symlink' is synonymous with 'absolute' links, which are different from 'relative' links. Although all of these are symbolic links, the nomenclature here is defined such that 'symlink' and 'absolute' are equivalent . When verifying a link, choosing 'relative' means that the link *must* be relative to the source, so relative and absolute links are mutually exclusive.

'force_update'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false force copy update always

Default value: false

Example:

```
body copy_from example
{
force_update => "true";
}
```

Notes:

Warning: this is a non-convergent operation. Although the end point might stabilize in content, the operation will never quiesce. Use of this feature is not recommended except in exceptional circumstances since it creates a busy-dependency. If the copy is a network copy, the system will be disturbed by network disruptions.

'force_ipv4'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false force use of ipv4 on ipv6 enabled network

Default value: false

Example:

```
body copy_from example
{
force_ipv4 => "true";
}
```

Notes:

IPv6 should be harmless to most users unless you have a partially or misconfigured setup.

'portnumber'

Type: int

Allowed input range: `1024,99999`

Synopsis: Port number to connect to on server host

Example:

```
body copy_from example
{
portnumber => "5308";
```

```
}
```

Notes:

The standard or registered port number is tcp/5308. CFEngine does not presently use its registered udp port with the same number, but this could change in the future.

'preserve' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to preserve file permissions on copied file

Default value: false

Example:

```
body copy_from example
{
preserve => "true";
}
```

Notes:

Ensures the destination file (promiser) gets the same Unix mode as the source. This also applies to remote copies.

History: Was introduced in version 3.1.0b3,Nova 2.0.0b1 (2010)

'purge' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false purge files on client that do not match files on server when a depth_search is used

Default value: false

Example:

```
body copy_from example
{
purge => "true";
}
```

Notes:

Purging files is a potentially dangerous matter during a file copy it implies that any promiser (destination) file which is not matched by a source will be deleted. Since there is no source, this means the file will be irretrievable. Great care should be exercised when using this feature.

Note that purging will also delete backup files generated during the file copying if copy_backup is set to true.

'stealth' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to preserve time stamps on copied file

Default value: false

Example:

```
body copy_from example
{
stealth => "true";
}
```

Notes:

Preserves file access and modification times on the promiser files.

'timeout' **Type**: int

Allowed input range: `1,3600`

Synopsis: Connection timeout, seconds

Example:

```
body runagent control
{
timeout => "10";
}
```

Notes:

Timeout in seconds.

'trustkey' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false trust public keys from remote server if previously unknown

Default value: false

Example:

```
body copy_from example
{
trustkey => "true";
}
```

Notes:

If the server's public key has not already been trusted, this allows us to accept the key in automated key-exchange.

Note that, as a simple security precaution, trustkey should normally be set to '`false`', to avoid key exchange with a server one is not one hundred percent sure about, though

the risks for a client are rather low. On the server-side however, trust is often granted to many clients or to a whole network in which possibly unauthorized parties might be able to obtain an IP address, thus the trust issue is most important on the server side.

As soon as a public key has been exchanged, the trust option has no effect. A machine that has been trusted remains trusted until its key is manually revoked by a system administrator. Keys are stored in 'WORKDIR/ppkeys'.

'type_check'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false compare file types before copying and require match

Example:

```
body copy_from example
{
type_check => "false";
}
```

Notes:

File types at source and destination should normally match in order for updates to overwrite them. This option allows this checking to be switched off.

'verify'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false verify transferred file by hashing after copy (resource penalty)

Default value: false

Example:

```
body copy_from example
{
verify => "true";
}
```

Notes:

This is a highly resource intensive option, not recommended for large file transfers.

7.4.4 create

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false whether to create non-existing file

Example:

```
files:

  "/path/plain_file"

     create =>   "true";

  "/path/dir/."

     create =>   "true";
```

Notes:

Directories are created by using the '/.' to signify a directory type. Note that, if no permissions are specified, mode 600 is chosen for a file, and mode 755 is chosen for a directory. If you cannot accept these defaults, you *should* specify permissions.

Note that technically, '/.' is a regular expression. However, it is used as a special case meaning "directory". See the **filenames and regular expressions** near the beginning of the section on Section 7.4 [files promises], page 218 for a more complete discussion.

Note: In general, you should not use create with Section 7.4.3 [copy_from], page 233 or Section 7.4.12 [link_from], page 267 in files promises. These latter attributes automatically create the promised file, and using create may actually prevent the copy or link promise from being kept (since create acts first, which may affect file comparison or linking operations).

7.4.5 delete (body template)

Type: (ext body)

'dirlinks' **Type**: (menu option)

Allowed input range:

```
delete
tidy
keep
```

Synopsis: Menu option policy for dealing with symbolic links to directories during deletion

Example:

```
body delete example
{
dirlinks => "keep";
}
```

Notes:

Links to directories are normally removed just like any other link or file objects. By keeping directory links, you preserve the logical directory structure of the file system so that a link to a directory is not removed but is treated as a directory to be descended into.

The value keep instructs CFEngine not to remove directory links. The values delete and tidy are synonymous, and instruct CFEngine to remove directory links.

Default value (only if body is present):

The default value only has significance if there is a delete body present. If there is no delete body, then files (and directory links) are **not** deleted.

dirlinks => delete

'rmdirs' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to delete empty directories during recursive deletion

Example:

```
body delete example
{
rmdirs => "true";
}
```

Notes:

When deleting files recursively from a base directory, should we delete empty directories also, or keep the directory structure intact?

Note the parent directory of a search is not deleted in recursive deletions. In CFEngine 2 there was an option to delete the parent of the search, but now in CFEngine 3, you must code a separate promise to delete the single parent object.

```
bundle agent cleanup
{
files:

  # This will not delete the parent

  "/home/mark/tmp/testcopy"

    delete => tidyfiles,
    file_select => changed_within_1_year,
    depth_search => recurse("inf");

  # Now delete the parent.

  "/home/mark/tmp/testcopy"
    delete => tidyfiles;
}
```

```
body delete tidyfiles
{
dirlinks => "delete";
rmdirs   => "true";
}

body file_select changed_within_1_year
{
mtime     => irange(ago(1,0,0,0,0,0),now);
file_result => "mtime";
}
```

Default value (only if body is present):

The default value only has significance if there is a `delete` body present. If there is no `delete` body, then files (and directories) are **not** deleted.

`rmdirs => true`

7.4.6 `depth_search` (body template)

Type: (ext body)

'`depth`'
Type: int

Allowed input range: `0,99999999999`

Synopsis: Maximum depth level for search

Example:

```
body depth_search example
{
depth => "inf";
}
```

Notes:

This was previous called 'recurse' in earlier versions of CFEngine. Note that the value '`inf`' may be used for an unlimited value.

When searching recursively from a directory, the parent directory is not part of the search. It is only the anchor point. To alter the parent, a separate non-recursive promise should be made.

'`exclude_dirs`'
Type: slist

Allowed input range: `.*`

Synopsis: List of regexes of directory names NOT to include in depth search

Example:

```
body depth_search
{
# no dot directories
exclude_dirs => { "\..*" };
}
```

Notes:

Directory names are treated specially when searching recursively through a file system.

'include_basedir'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false include the start/root dir of the search results

Example:

```
body depth_search example
{
include_basedir => "true";
}
```

Notes:

When checking files recursively (with `depth_search`) the promiser is a directory. This parameter determines whether that initial directory should be considered part of the promise or simply a boundary which marks the edge of the search. If true, the promiser directory will also promise the same attributes as the files inside it.

'include_dirs'

Type: slist

Allowed input range: .*

Synopsis: List of regexes of directory names to include in depth search

Example:

```
body depth_search example
{
include_dirs => { "subdir1", "subdir2", "pattern.*" };
}
```

Notes:

This is the complement of exclude_dirs.

'rmdeadlinks'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false remove links that point to nowhere

Default value: false

Example:

```
body depth_search example
{
rmdeadlinks => "true";
}
```

Notes:

If we find links that point to non-existence files, should we delete them or keep them?

'traverse_links'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false traverse symbolic links to directories

Default value: false

Example:

```
body depth_search example
{
traverse_links => "true";
}
```

Notes:

If this is true, `cf-agent` will treat symbolic links to directories as if they were directories. Normally this is considered a potentially dangerous assumption and links are not traversed.

'`xdev`' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false exclude directories that are on different devices

Default value: false

Example:

```
body depth_search example
{
xdev => "true";
}
```

Notes:

7.4.7 edit_defaults (body template)

Type: (ext body)

'edit_backup'

Type: (menu option)

Allowed input range:

```
true
false
timestamp
rotate
```

Synopsis: Menu option for backup policy on edit changes

Default value: true

Example:

```
body edit_defaults example
{
edit_backup => "timestamp";
}
```

Notes:

'empty_file_before_editing'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Baseline memory model of file to zero/empty before commencing promised edits

Default value: false

Example:

```
body edit_defaults example
{
empty_file_before_editing => "true";
}
```

Notes:

Emptying a file before reconstructing its contents according to a fixed recipe allows an ordered procedure to be convergent.

'inherit' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: If true this causes the sub-bundle to inherit the private classes of its parent

Example:

```
bundle agent name
{
methods:

  "group name" usebundle => my_method,
                 inherit => "true";
}


body edit_defaults example
{
inherit => "true";
}
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

Default value: false

The `inherit` constraint can be added to the CFEngine code in two places: for `edit_defaults` and in `methods` promises. If set to true, it causes the child-bundle named in

the promise to inherit (only) the classes of the parent bundle. Inheriting the variables is unnecessary as the child can always access the parent's variables by a qualified reference using its bundle name, e.g. '$(bundle.variable)'.

'max_file_size'

Type: int

Allowed input range: 0,99999999999

Synopsis: Do not edit files bigger than this number of bytes

Example:

```
body edit_defaults example
{
max_file_size => "50K";
}
```

Notes:

A local, per-file sanity check to make sure the file editing is sensible. If this is set to zero, the check is disabled and any size may be edited. The default value of max_file_size is determined by the global control body setting, See Section 5.2.17 [editfilesize in agent], page 80, whose default value is 100k.

'recognize_join'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Join together lines that end with a backslash, up to 4kB limit

Default value: false

Example:

```
files:

  "/tmp/test_insert"
            create => "true",
         edit_line => Insert("$(insert.v)"),
```

```
     edit_defaults => join;
}

#

body edit_defaults join
{
recognize_join => "true";
}
```

Notes:

If set to true, this option allows CFEngine to process line based files with backslash continuation. The default is to not process continuation backslashes.

Back slash lines will only be concatenated if the file requires editing, and will not be restored. Restoration of the backslashes is not possible in a meaningful and convergent fashion.

'rotate' **Type**: int

Allowed input range: `0,99`

Synopsis: How many backups to store if 'rotate' edit_backup strategy is selected. Defaults to 1

Example:

```
body rename example
{
rotate => "4";
}
```

Notes:

Used for log rotation. If the file is named '`foo`' and the '`rotate`' attribute is set to '`4`', as above, then initially '`foo`' is copied to '`foo.1`' and the old file '`foo`' is zeroed out (that is, the inode of the original logfile does not change, but the original logfile will be empty after the rotation is complete).

The next time the promise is executed, '`foo.1`' will be renamed '`foo.2`', '`foo`' is again copied to '`foo.1`' and the old file '`foo`' is again zeroed out.

Each time the promise is executed (and typically, the promise would be executed as guarded by time-based or file-size-based classes), the files are copied/zeroed or rotated as above until there are '`rotate`' numbered files plus the one "main" file. In the example above, the file '`foo.3`' will be renamed '`foo.4`', but the old version of the file '`foo.4`' will be deleted (that is, it "falls off the end" of the rotation).

7.4.8 edit_line

Type: (ext bundle) (Separate Bundle)

7.4.9 edit_template

Type: string
Allowed input range: "?(/.*)
Synopsis: The name of a special CFEngine template file to expand
Example:

```
#This is a template file /templates/input.tmpl

These lines apply to anyone

[%CFEngine solaris.Monday:: %]
Everything after here applies only to solaris on Mondays
until overridden...

[%CFEngine linux:: %]
Everything after here now applies now to linux only.

[%CFEngine BEGIN %]
This is a block of text
That contains list variables: $(some.list)
With text before and after.
[%CFEngine END %]

nameserver $(some.list)
```

For example:

```
[%CFEngine any:: %]
<VirtualHost $(sys.ipv4[eth0]):80>
        ServerAdmin             $(stage_file.params[apache_mail_address][1])
        DocumentRoot            /var/www/htdocs
        ServerName              $(stage_file.params[apache_server_name][1])
        AddHandler              cgi-script cgi
        ErrorLog                /var/log/httpd/error.log
        AddType                 application/x-x509-ca-cert .crt
        AddType                 application/x-pkcs7-crl    .crl
        SSLEngine               off
        CustomLog               /var/log/httpd/access.log
</VirtualHost>

[%CFEngine webservers_prod:: %]
[%CFEngine BEGIN %]
<VirtualHost $(sys.ipv4[$(bundle.interfaces)]):443>
        ServerAdmin             $(stage_file.params[apache_mail_address][1])
```

```
        DocumentRoot            /var/www/htdocs
        ServerName              $(stage_file.params[apache_server_name][1])
        AddHandler              cgi-script cgi
        ErrorLog                /var/log/httpd/error.log
        AddType                 application/x-x509-ca-cert .crt
        AddType                 application/x-pkcs7-crl    .crl
        SSLEngine               on
        SSLCertificateFile      $(stage_file.params[apache_ssl_crt][1])
        SSLCertificateKeyFile   $(stage_file.params[apache_ssl_key][1])
        CustomLog               /var/log/httpd/access.log
</VirtualHost>
[%CFEngine END %]
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

The template format uses inline tags to mark regions and classes. Each line represents an `insert_lines` promise, unless the promises are grouped into a block using:

```
[%CFEngine BEGIN %]
...
[%CFEngine END %]
```

Variables, scalars and list variables are expanded within each promise; so, if lines are grouped into a block, the whole block is repeated when lists are expanded (see the Special Topics Guide on editing).

If a class-context modified is used:

```
[%CFEngine class-expression:: %]
```

then the lines that follow are only inserted if the context matches the agent's current context. This allows conditional insertion.

7.4.10 `edit_xml`

Type: (ext bundle) (Separate Bundle)

7.4.11 `file_select` (body template)

Type: (ext body)

'`leaf_name`'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of regexes that match an acceptable name

Example:

```
body file_select example
{
leaf_name => { "S[0-9]+[a-zA-Z]+", "K[0-9]+[a-zA-Z]+" };
```

```
file_result => "leaf_name";
}
```

Notes:

This pattern matches only the node name of the file, not its path.

'path_name'

Type: slist

Allowed input range: `"?(/.*)`

Synopsis: List of pathnames to match acceptable target

Example:

```
body file_select example
{
leaf_name => { "prog.pid", "prog.log" };
path_name => { "/etc/.*", "/var/run/.*" };

file_result => "leaf_name.path_name"
}
```

Notes:

Path name and leaf name can be conveniently tested for separately by use of appropriate regular expressions.

'search_mode'

Type: slist

Allowed input range: `[0-7augorwxst,+-]+`

Synopsis: A list of mode masks for acceptable file permissions

Example:

```
#######################################################
#
# Searching for permissions
#
#######################################################

body common control
```

```
   {
   any::

      bundlesequence  => {
                         "testbundle"
                         };

   version => "1.2.3";
   }

#############################################

bundle agent testbundle

{
files:

  "/home/mark/tmp/testcopy"

    file_select => by_modes,
    transformer => "/bin/echo DETECTED $(this.promiser)",
    depth_search => recurse("inf");

}

#############################################

body file_select by_modes

{
search_mode => { "711" , "666" };
file_result => "mode";
}

#############################################

body depth_search recurse(d)

{
depth => "$(d)";
}
```

Notes:

The mode may be specified in symbolic or numerical form with '+' and '-' constraints. Note that concatenation ug+s implies u OR g, and u+g,u+s implies u AND g.

'search_size'

Type: irange [int,int]

Allowed input range: 0,inf

Synopsis: Integer range of file sizes

Example:

```
body file_select example
{
search_size => irange("0","20k");
file_result => "size";
}
```

Notes:

'search_owners'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of acceptable user names or ids for the file, or regexes to match

Example:

```
body file_select example
{
search_owners => { "mark", "jeang", "student_.*" };
file_result => "owner";
}
```

Notes:

A list of regular expressions any of which must match the entire userid, (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). Note that windows does not have user ids, only names.

'search_groups'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of acceptable group names or ids for the file, or regexes to match

Example:

```
body file_select example
{
search_groups => { "users", "special_.*" };
file_result => "group";
}
```

Notes:

A list of regular expressions, any of which which must match the entire group, (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). Note that on windows, files do not have group associations.

'search_bsdflags'

Type: slist

Allowed input range: `[+-]*[(arch|archived|nodump|opaque|sappnd|sappend|schg|schange|simmutab`

Synopsis: String of flags for bsd file system flags expected set

Example:

```
body file_select xyz
{
search_bsdflags => "archived|dump";
file_result => "bsdflags";
}
```

Notes:

Extra BSD file system flags (these have no effect on non-BSD versions of CFEngine). See the manual page for `chflags` for more details.

'ctime'

Type: irange [int,int]

Allowed input range: `0,2147483647`

Synopsis: Range of change times (ctime) for acceptable files

Example:

```
body files_select example
{
```

```
ctime => irange(ago(1,0,0,0,0,0),now);
file_result => "ctime";
}
```

Notes:

The file's change time refers to both modification of content and attributes such as permissions. On windows, `ctime` refers to creation time.

'mtime'

Type: irange [int,int]

Allowed input range: `0,2147483647`

Synopsis: Range of modification times (mtime) for acceptable files

Example:

```
body files_select example

{
# Files modified more than one year ago (i.e., not in mtime range)
mtime => irange(ago(1,0,0,0,0,0),now);
file_result => "!mtime";
}
```

Notes:

The file's modification time refers to both modification of content but not other attributes such as permissions.

'atime'

Type: irange [int,int]

Allowed input range: `0,2147483647`

Synopsis: Range of access times (atime) for acceptable files

Example:

```
body file_select used_recently
{

# files accessed within the last hour
atime     => irange(ago(0,0,0,1,0,0),now);
file_result => "atime";
}
```

```
body file_select not_used_much

{
# files not accessed since 00:00 1st Jan 2000 (in the local timezime)
atime     => irange(on(2000,1,1,0,0,0),now);
file_result => "!atime";
}
```

Notes:

A range of times during which a file was accessed can be specified in a `file_select` body. (Like file filters in CFEngine 2.)

'`exec_regex`'

Type: string

Allowed input range: `.*`

Synopsis: Matches file if this regular expression matches any full line returned by the command

Example:

```
body file_select example
{
exec_regex => "SPECIAL_LINE: .*";
exec_program => "/path/test_program $(this.promiser)";
file_result => "exec_program.exec_regex";
}
```

Notes:

The regular expression must be used in conjunction with the `exec_program` test. In this way the program must both return exit status 0 and its output must match the regular expression. The entire output must be matched (that is, as if the regex is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'`exec_program`'

Type: string

Allowed input range: `"?(/.*)`

Synopsis: Execute this command on each file and match if the exit status is zero

Example:

```
body file_select example
{
exec_program => "/path/test_program $(this.promiser)";
file_result => "exec_program";
}
```

Notes:

This is part of the customizable file search criteria. If the user-defined program returns exit status 0, the file is considered matched.

'file_types'

Type: (option list)

Allowed input range:

```
plain
reg
symlink
dir
socket
fifo
door
char
block
```

Synopsis: List of acceptable file types from menu choices

Example:

```
body file_select filter
{
file_types => { "plain","symlink" };

file_result => "file_types";
}
```

Notes:

File types vary in details between operating systems. The main POSIX types are provided here as menu options, with 'reg' being a synonym for 'plain' (meaning, in both cases, not one of the "special" file types).

'issymlinkto'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of regular expressions to match file objects

Example:

```
body file_select example
{
issymlinkto => { "/etc/[^/]*", "/etc/init\.d/[a-z0-9]*" };
}
```

Notes:

A list of regular expressions. If the file is a symbolic link which points to files matched by one of these expressions, the file will be selected. As windows does not support symbolic links, this attribute is not applicable there.

'file_result'

Type: string

Allowed input range: `[!*(leaf_name|path_name|file_types|mode|size|owner|group|atime|ctime|mt regex|exec_program|bsdflags)[|&.]*]*`

Synopsis: Logical expression combining classes defined by file search criteria

Example:

```
body file_select year_or_less

{
mtime       => irange(ago(1,0,0,0,0,0),now);
file_result => "mtime";
}

body file_select my_pdf_files_morethan1dayold

{
mtime         => irange(ago(0,0,1,0,0,0),now);
leaf_name     => { ".*\.pdf" , ".*\.fdf" };
search_owners => { "mark" };

file_result => "owner.leaf_name.!mtime";
}
```

Notes:

Sets the criteria for file selection outcome during file searches. The syntax is the same as for a class expression, since the file selection is a classification of the file-search in the same way that system classes are a classification of the abstract host-search (that is, you may specify a boolean expression involving any of the file-matching components). In this way, you may specify arbitrarily complex file-matching parameters, such as what is shown above, "is owned by mark, has the extension '.pdf' or '.fdf', and whose modification time is not between 1 day ago and now (that is, it is older than 1 day)".

Items in the boolean expression in `file_result` must be from the following list:

- leaf_name
- path_name
- file_types
- mode
- size
- owner
- group
- atime
- ctime
- mtime
- issymlinkto
- exec_regex
- exec_program
- bsdflags

7.4.12 `link_from` (body template)

Type: (ext body)

`copy_patterns`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A set of patterns that should be copied and synchronized instead of linked

Example:

```
body link_from example
{
copy_patterns =>  { "special_node1", "/path/special_node2" };
}
```

Notes:

During the linking of files, it is sometimes useful to buffer changes with an actual copy, especially if the link is to an emphemeral file system. This list of patterns matches files that arise during a linking policy. A positive match means that the file should be copied and updated by modification time.

'link_children'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to link all directory's children to source originals

Default value: false

Example:

```
body link_from example
{
link_children => "true";
}
```

Notes:

If the promiser is a directory, instead of copying the children, link them to the source.

'link_type'

Type: (menu option)

Allowed input range:

```
symlink
hardlink
relative
absolute
```

Synopsis: The type of link used to alias the file

Default value: symlink

Example:

```
body link_from example
{
link_type => "symlink";
source => "/tmp/source";
}
```

Notes:

What kind of link should be used to link files. Users are advised to be wary of 'hard links' (see Unix manual pages for the 'ln' command). The behaviour of non-symbolic links is often precarious and unpredictable. However, hard links are the only supported type by windows.

Note that 'symlink' is synonymous with 'absolute' links, which are different from 'relative' links. Although all of these are symbolic links, the nomenclature here is defined such that 'symlink' and 'absolute' are equivalent . When verifying a link, choosing 'relative' means that the link *must* be relative to the source, so relative and absolute links are mutually exclusive.

'source' **Type**: string

Allowed input range: .+

Synopsis: The source file to which the link should point

Example:

```
body copy_from example
{
source => "/path/to/source";
}

# or

body link_from example
{
source => "/path/to/source";
}
```

Notes:

For remote copies this refers to the file name on the remote server.

'when_linking_children'

Type: (menu option)

Allowed input range:

```
override_file
if_no_such_file
```

Synopsis: Policy for overriding existing files when linking directories of children

Example:

```
body link_from example
{
when_linking_children => "if_no_such_file";
}
```

Notes:

The options refer to what happens if the directory exists already and is already partially populated with files. If the directory being copied from contains a file with the same name as that of a link to be created, we must decide whether to override the existing destination object with a link or simply omit the automatic linkage for files that already exist. The latter case can be used to make a copy of one directory with certain fields overridden.

'when_no_source'

Type: (menu option)

Allowed input range:

```
force
delete
nop
```

Synopsis: Behaviour when the source file to link to does not exist

Default value: nop

Example:

```
body link_from example
{
when_no_source => "force";
}
```

Notes:

If we try to create a link to a file that does not exist a link, how should CFEngine respond? The options are to force the creation to a file that does not (yet) exist, delete any existing link, or do nothing.

7.4.13 move_obstructions

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false whether to move obstructions to file-object creation

Example:

```
files:

  "/tmp/testcopy"

    copy_from    => mycopy("/tmp/source"),
    move_obstructions => "true",
    depth_search => recurse("inf");
```

Notes:

If we have promised to make file 'X' a link, but it already exists as a file, or vice-versa, or if a file is blocking the creation of a directory etc, then normally CFEngine will report an error. If this is set, existing objects will be moved aside to allow the system to heal without intervention. Files and directories are saved/renamed, but symbolic links are deleted.

Note that symbolic links for directories are treated as directories, not links. This behaviour can be discussed, but the aim is to err on the side of caution. Some operating systems (Solaris) use symbolic links in path names. Copying to a directory could then result in renaming of the important link, if the behaviour were different.

7.4.14 pathtype

Type: (menu option)

Allowed input range:

```
literal
regex
guess
```

Synopsis: Menu option for interpreting promiser file object

Example:

```
files:

   "/var/lib\d"
      pathtype => "guess",         # best guess (default)
         perms => system;

   "/var/lib\d"
      pathtype => "regex",# force regex interpretation
         perms => system;

   "/var/.*/lib"

      pathtype => "literal",       # force literal interpretation
         perms => system;
```

Notes:

By default, CFEngine makes an educated guess as to whether the promise pathname involves a regular expression or not. This guesswork is needed due to cross-platform differences in filename interpretation.

If CFEngine guesses (or is told) that the pathname uses a regular expression pattern, it will undertake a file search to find possible matches. This can consume significant resources, and so the '`guess`' option will always try to optimize this. Guesswork is, however, imperfect, so you have the option to declare your intention.

If the keyword `literal` is invoked, a path will be treated as a literal string regardless of what characters is contains. If it is declared '`regex`', it will be treated as a pattern to match.

Note that CFEngine splits the promiser up into path links before matching, so that each link in the path chain is matched separately. Thus it it meaningless to have a '/' in a regular expression, as the comparison will never see this character.

In the examples above, at least one case implies an iteration over all files/directories matching the regular expression, while the last case means a single literal object with a name composed of dots and stars.

Furthermore, on Windows paths using `regex` must use the forward slash (/) as path separator, since the backward slash has a special meaning in a regular expression. Literal paths may also use backslash (\) as a path separator, See Section 2.11.3 [Regular expressions in paths], page 37, for more information.

7.4.15 perms (body template)

Type: (ext body)

'bsdflags' **Type**: slist

Allowed input range: [+-]*[(arch|archived|nodump|opaque|sappnd|sappend|schg|schange|simmutab

Synopsis: List of menu options for bsd file system flags to set

Example:

```
body perms example

{
bsdflags => { "uappnd","uchg","uunlnk","nodump",
              "opaque","sappnd","schg","sunlnk" };
}
```

Notes:

The BSD Unices (FreeBSD, OpenBSD, NetBSD) and MacOSX have additional filesystem flags which can be set. Refer to the BSD chflags documentation for this.

'groups' **Type**: slist

Allowed input range: [a-zA-Z0-9_$.-]+

Synopsis: List of acceptable groups of group ids, first is change target

Example:

```
body perms example
{
groups => { "users", "administrators" };
}
```

Notes:

The first named group is the list is the defaul that will be configured if the file does not match an element of the list. The reserved word 'none' may be used to match files that are not owned by a registered group. On windows, files do not have file groups associated with them and thus this attribute is ignored.

ACLs may be used in place for this.

'mode' **Type**: string

Allowed input range: [0-7augorwxst,+-]+

Synopsis: File permissions (like posix chmod)

Example:

```
body perms example
{
mode => "a+rx,o+w";
}
```

Notes:

The mode string may be symbolic or numerical, like `chmod`. This is ignored on windows, as the permission model uses ACLs. ACLs are supported by CFEngine Nova.

'owners' **Type**: slist

Allowed input range: `[a-zA-Z0-9_$.-]+`

Synopsis: List of acceptable owners or user ids, first is change target

Example:

```
body perms example
{
owners => { "mark", "wwwrun", "jeang" };
}
```

Notes:

The first user is the reference value that CFEngine will set the file to if none of the list items matches the true state of the file. The reserved word '`none`' may be used to match files that are not owned by a registered user.

On windows, users can only take ownership of files, never give it. Thus, the first user in the list should be the user running the CFEngine process (usually "Administrator"). Additionally, some groups may be owners on windows (such as the "Administrators" group).

'rxdirs' **Type**: (menu option)

Allowed input range:

```
true
false
yes
```

```
no
on
off
```

Synopsis: true/false add execute flag for directories if read flag is set

Example:

```
body perms rxdirs
{
rxdirs => "false";
}
```

Notes:

Default behaviour is to set the 'x' flag on directories automatically if the 'r' flag is specified when specifying multiple files in a single promise. This is ignored on windows, as the permission model uses ACLs.

7.4.16 rename (body template)

Type: (ext body)

'disable' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false automatically rename and remove permissions

Default value: false

Example:

```
body rename example
{
disable => "true";
disable_suffix => ".nuked";
}
```

Notes:

Disabling a file means making is impotent in the context in which it has an effect. For executables this means preventing execution, for an information file it means making the file unreadable.

'disable_mode'

Type: string

Allowed input range: [0-7augorwxst,+-]+

Synopsis: The permissions to set when a file is disabled

Example:

```
body rename example
{
disable_mode => "0600";
}
```

Notes:

To disable an executable it is not enough to rename it, you should also remove the executable flag.

'disable_suffix'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The suffix to add to files when disabling (.cfdisabled)

Example:

```
body rename example
{
disable => "true";
disable_suffix => ".nuked";
}
```

Notes:

To make disabled files in a particular manner, use this string suffix. The default value is '.cf-disabled'.

'newname' **Type**: string

Allowed input range: (arbitrary string)

Synopsis: The desired name for the current file

Example:

```
body rename example(s)
{
newname => "$(s)";
}
```

Notes:

'rotate' **Type**: int

Allowed input range: 0,99

Synopsis: Maximum number of file rotations to keep

Example:

```
body rename example
{
rotate => "4";
}
```

Notes:

Used for log rotation. If the file is named 'foo' and the 'rotate' attribute is set to '4', as above, then initially 'foo' is copied to 'foo.1' and the old file 'foo' is zeroed out (that is, the inode of the original logfile does not change, but the original logfile will be empty after the rotation is complete).

The next time the promise is executed, 'foo.1' will be renamed 'foo.2', 'foo' is again copied to 'foo.1' and the old file 'foo' is again zeroed out.

Each time the promise is executed (and typically, the promise would be executed as guarded by time-based or file-size-based classes), the files are copied/zeroed or rotated as above until there are 'rotate' numbered files plus the one "main" file. In the example above, the file 'foo.3' will be renamed 'foo.4', but the old version of the file 'foo.4' will be deleted (that is, it "falls off the end" of the rotation).

7.4.17 repository

Type: string

Allowed input range: `"?(/.*)`

Synopsis: Name of a repository for versioning

Example:

```
files:

 "/path/file"

   copy_from => source,
   repository => "/var/cfengine/repository";
```

Notes:

A local repository for this object, overrides the default, See Section 5.2.33 [default_repository], page 89.

Note that when a repository is specified, the files are stored using the canonified directory name of the original file, concatenated with the name of the file. So, for example, '/usr/local/etc/postfix.conf' would ordinarily be stored in an alternative repository as '_usr_local_etc_postfix.conf.cfsaved'.

7.4.18 touch

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to touch time stamps on file

Example:

```
files:

 "/path/file"

   touch => "true";
```

Notes:

7.4.19 transformer

Type: string
Allowed input range: "?(/.*)
Synopsis: Command (with full path) used to transform current file (no shell wrapper used)
Example:

```
files:
  "/home/mark/tmp/testcopy"

    file_select => pdf_files,
    transformer => "/usr/bin/gzip $(this.promiser)",
    depth_search => recurse("inf");


 classes:
    "do_update" expression => isnewerthan("/etc/postfix/alias",
                                          "/etc/postfix/alias.cdb");

 files:
    "/etc/postfix/alias.cdb"
       create => "true",           # Must have this!
       transformer => "/usr/sbin/postalias /etc/postfix/alias",
       ifvarclass => "do_update";
```

Notes:

A command to execute, usually for the promised file to transform it to something else (but possibly to create the promised file based on a different origin file). The examples above show both types of promises.

The promiser file must exist in order to effect the transformer.

In the first example, the promise is made on the file that we wish to transform. If the promised file exists, the transformer will change the file to a compressed version (and the next time CFEngine runs, the promised file will no longer exist, because it now has the '.gz' extension).

In the second example, the promise is made on the file *resulting from* the transformation (and the promise is conditional on the original file being newer than the result file). In this case, we *must* specify 'create => true'. If we do not, then if the promised file is removed, the transformer will not be executed.

Note also that if you use the $(this.promiser) variable or other variable in this command, and the file object contains spaces you should quote the variable, e.g.

```
transformer => "/usr/bin/gzip \"$(this.promiser)\"",
```

Note also that the transformer does not actually need to change the file. You can, for example, simply report on the existence of files with

```
transformer => "/bin/echo I found a file named $(this.promiser)",
```

The file streams `stdout` and `stderr` are redirected by CFEngine, and will not appear in any output unless you run `cf-agent` with the '`-v`' switch (or enable `verbose` in an `outputs` promise).

It is possible to set classes based on the return code of a transformer-command in in a very flexible way. See the `kept_returncodes`, `repaired_returncodes` and `failed_returncodes` attributes.

Finally, you should note that the command is not run in a shell - which means that you cannot perform file redirection or create pipelines.

7.5 Miscelleneous in `edit_line` promises

Line based editing is a simple model for editing files. Before XML, and later JSON, most configuration files were line based. The line-based editing offers a powerful environment for model-based editing and templating.

7.5.1 `select_region` (body template)

Type: (ext body)

'`include_start_delimiter`'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Whether to include the section delimiter

Default value: false

Example:

```
body select_region MySection(x)
{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
include_start_delimiter => "true";
}
```

Notes:

In a sectioned file, the line that marks the opening of a section is not normally included in the defined region (that is, it is recognized as a delimiter, but it is not included as one of the lines available for editing). Setting this option to true makes it included. e.g. in this example

```
[My section]
one
two
three
```

the section does not normally include the line '[My section]'. By setting include_start_delimiter to 'true'it would be possible for example, to delete the entire section, including the section header. If however include_start_delimiter is 'false', the *contents* of the section could be deleted, but the header would be unaffected by any delete_lines promises. See the next section on include_start_delimiter for further details.

History: This attribute was introduced in CFEngine version 3.0.5 (2010)

'include_end_delimiter'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Whether to include the section delimiter

Default value: false

Example:

```
body select_region BracketSection(x)
{
select_start => "$(x) \{";
select_end => "}";
include_end_delimiter => "true";
}
```

Notes:

In a sectioned file, the line that marks the end of a section is not normally included in the defined region (that is, it is recognized as a delimiter, but it is not included as one of the lines available for editing). Setting this option to true makes it included. e.g. in this example

```
/var/log/mail.log {
    monthly
    missingok
    notifempty
    rotate 7
    }
```

the section does not normally include the line containing '}'. By setting `include_end_delimiter` to '`true`'it would be possible for example, to delete the entire section, including the section trailer. If however `include_end_delimiter` is '`false`', the *contents* of the section could be deleted, but the header would be unaffected by any `delete_lines` promises.

The use of `include_start_delimiter` and `include_end_delimiter` depend on the type of sections you are dealing with, and what you want to do with them. Note that sections can be bounded at both the start and end (as in the example above) or just at the start (as in the sample shown in `include_start_delimiter`).

History: This attribute was introduced in CFEngine version 3.0.5 (2010)

'`select_start`'

Type: string

Allowed input range: `.*`

Synopsis: Regular expression matching start of edit region

Example:

```
body select_region example(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}
```

Notes:

See also `select_end`. These delimiters mark out the region of a file to be edited. In the example, it is assumed that the file has section markers.

```
[section 1]

lines.
lines...
```

```
[section 2]

lines ....
etc..
```

The start marker includes the first matched line.

'select_end'

Type: string

Allowed input range: .*

Synopsis: Regular expression matches end of edit region from start

Example:

```
body select_region example(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}
```

Notes:

See also select_start. These delimiters mark out the region of a file to be edited. In the example, it is assumed that the file has section markes

```
[section 1]

lines.
lines...


[section 2]

lines ....
etc..
```

If you want to match from a starting location to the end of the file (even if there are other lines matching select_start intervening), then just omit the select_end promise and the selected region will run to the end of the file.

7.6 delete_lines promises in 'edit_line'

This promise assures that certain lines matching regular expression patterns exactly will not be present in a text file. If the lines are found, the default promise is to remove them (this behavior may be modified with further pattern matching in delete_select and/or changed with not_matching).

```
bundle edit_line example
  {
  delete_lines:

    "olduser:.*";

  }
```

Note that typically, only a single line is specified in each `delete_lines` promise (you may of course have multiple promises that each delete a line).

It is also possible to specify multi-line `delete_lines` promises. However, these promises will only delete those lines if *all* the lines are present in the file *in exactly the same order* as specified in the promise (with no intervening lines). That is, all the lines must match as a unit for the `delete_lines` promise to be kept.

If the promiser is contains multiple lines, then CFEngine assumes that all of the lines must exist as a contiguous block in order to be deletes. This gives '`preserve_block`' semantics to any multiline `delete_lines` promise.

7.6.1 `delete_select` (body template)

Type: (ext body)

'`delete_if_startwith_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if it starts with a string in the list

Example:

```
body delete_select example(s)
{
delete_if_startwith_from_list => { @(s) };
}
```

Notes:

Delete lines from a file if they begin with the sub-strings listed. Note that this determination is made only on promised lines (that is, this attribute modifies the selection criteria, it does not make the initial selection). Therefore, if the file contains the following lines:

```
start alpha igniter
start beta igniter
init alpha burner
init beta burner
stop beta igniter
stop alpha igniter
stop alpha burner
```

Then the following promise initially selects the four lines containing 'alpha', but is moderated by the delete_select attribute. Thus, the promise will delete only the first and third lines of the file:

```
bundle edit_line alpha
{
delete_lines:
    ".*alpha.*"
         delete_select => starters;
}

body delete_select starters
{
    delete_if_startwith_from_list => { "begin", "start", "init" };
}
```

'delete_if_not_startwith_from_list'

Type: slist

Allowed input range: .*

Synopsis: Delete line if it DOES NOT start with a string in the list

Example:

```
body delete_select example(s)
{
delete_if_not_startwith_from_list => { @(s) };
}
```

Notes:

Delete lines from a file unless they start with the sub-strings in the list given. Note that this determination is made only on promised lines (that is, this attribute modifies the selection criteria, it does not make the initial selection).

'delete_if_match_from_list'

Type: slist

Allowed input range: .*

Synopsis: Delete line if it fully matches a regex in the list

Example:

```
body delete_select example(s)
{
delete_if_match_from_list => { @(s) };
}
```

Notes:

Delete lines from a file if the lines *completely* match any of the regular expressions listed (that is, the regular expression is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

Note that the "match" determination is made only on promised lines (that is, this attribute modifies the selection criteria, it does not make the initial selection).

'`delete_if_not_match_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if it DOES NOT fully match a regex in the list

Example:

```
body delete_select example(s)
{
delete_if_not_match_from_list => { @(s) };
}
```

Notes:

Delete lines from a file unless the lines *completely* match any of the regular expressions listed (that is, the regular expressions are anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

Note that the "match" determination is made only on promised lines (that is, this attribute modifies the selection criteria, it does not make the initial selection).

'`delete_if_contains_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if a regex in the list match a line fragment

Example:

```
body delete_select example(s)
{
delete_if_contains_from_list => { @(s) };
}
```

Notes:

Delete lines from a file if they contain the sub-strings listed. Note that this determination is made only on promised lines (that is, this attribute modifies the selection criteria, it does not make the initial selection).

'delete_if_not_contains_from_list'

Type: slist

Allowed input range: .*

Synopsis: Delete line if a regex in the list DOES NOT match a line fragment

Example:

```
body delete_select discard(s)
{
delete_if_not_contains_from_list => { "substring1", "substring2" };
}
```

Notes:

Delete lines from the file which do not contain the sub-strings listed. Note that this determination is made only on promised lines (that is, this attribute modifies the selection criteria, it does not make the initial selection).

7.6.2 not_matching

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
```

```
off
```

Default value: false
Synopsis: true/false negate match criterion
Example:

```
delete_lines:

  # edit /etc/passwd - account names that are not "mark" or "root"

  "(mark|root):.*" not_matching => "true";
```

Notes:

When this option is true, it negates the pattern match of the promised lines (for convenience). **NOTE** that this does not negate any condition expressed in `delete_select` - it only negates the match of the initially promised lines.

Note, this makes no sense for multi-line deletions and is therefore disallowed. Either a multi-line promiser matches and it should be removed (i.e. `not_matching` is false) or it doesn't match the whole thing and the ordered lines have no meaning anymore as an entity. In this case, the lines can be separately stated.

7.7 `insert_lines` promises in '`edit_line`'

This promise is part of the line-editing model. It inserts lines into the file at a specified location. The location is determined by body-attributes. The promise object referred to can be a literal line of a file-reference from which to read lines.

```
insert_lines:

  "literal line or file reference"

     location => location_body,
     ...;
```

```
body common control

{
any::

  bundlesequence  => {
```

```
                     example
                     };
}

#######################################################

bundle agent example

{
files:

  "/var/spool/cron/crontabs/root"

     edit_line => addline;
}

#######################################################
# For the library
#######################################################

bundle edit_line addline

{
insert_lines:

 "0,5,10,15,20,25,30,35,40,45,50,55 * * * * /var/cfengine/bin/cf-execd -F";

}
```

By parameterizing the editing bundle, one can make generic and reusable editing bundles.

Note, when inserting multiple lines anchored to a particular place in a file, be careful with your intuition. If your intention is to insert a set of lines in a given order after a marker, then the following is incorrect:

```
bundle edit_line x
{
insert_lines:

  "line one" location => myloc;
  "line two" location => myloc;
}

body location myloc
```

```
{
select_line_matching => "# Right here.*";
before_after => "after";
}
```

This will reverse the order of the lines and will not converge, since the anchoring after the marker applies independently for each new line. This is not a bug, but an error of logic.

What was probably intended was to add multiple ordered lines after the marker, which should be a single correlated promise.

```
bundle edit_line x
{
insert_lines:

 "line one$(const.n)line two" location => myloc;

}
```

Or:

```
bundle edit_line x
{
insert_lines:

  "line one
line two" location => myloc;

}
```

7.7.1 expand_scalars

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: Expand any unexpanded variables

Example:

```
body common control

{
bundlesequence  => { "testbundle"  };
}

########################################################

bundle agent testbundle

{
files:

  "/home/mark/tmp/file_based_on_template"

       create    => "true",
       edit_line => ExpandMeFrom("/tmp/source_template");


}

########################################################

bundle edit_line ExpandMeFrom(template)
{
insert_lines:

   "$(template)"

          insert_type => "file",
       expand_scalars => "true";
}
```

Notes:

A way of incorporating templates with variable expansion into file operations. Variables should be named and scoped appropriately for the bundle in which this promise is made. i.e. you should qualify the variables with the bundle in which they are defined:

```
$(bundle.variable)
$(sys.host)
$(mon.www_in)
```

In CFEngine 2 `editfiles` this was called '`ExpandVariables`'.

7.7.2 insert_type

Type: (menu option)

Allowed input range:

```
literal
string
file
file_preserve_block
preserve_block
```

Default value: literal

Synopsis: Type of object the promiser string refers to

Example:

```
bundle edit_line lynryd_skynyrd
{
 vars:
    "keepers" slist => { "Won't you give me", "Gimme three steps" };

 insert_lines:

     "And you'll never see me no more"
       insert_type => "literal";   # the default

     "/song/lyrics"
       insert_type => "file",               # read selected lines from /song/lyrics
       insert_select => keep("@{keepers}");
}

body insert_select keep(s)
{
insert_if_startwith_from_list => { "@(s)" };
}
```

This will ensure that the following lines are inserted into the promised file:

```
And you'll never see me no more
Gimme three steps, Mister
Gimme three steps towards the door
Gimme three steps
```

Notes:

The default is to treat the promiser as a literal string of convergent lines (the values `literal` and `string` are synonymonous).

The default behaviour assumes that multi-line entries are not ordered specifically, and should be treated as a collection of lines of text and not as a single unbroken object.

If the option 'preserve_block' is used, then CFEngine will not break up multiple lines into individual, non-ordered objects, so that the block of text will be preserved. Even if some of the lines in the block already exist, they will be added again as a coherent block. Thus if you suspect that some stray / conflicting lines might be present they should be cleaned up with delete_lines first.

The value file is used to tell CFEngine that the string is non-literal and should be interpreted as a filename from which to import lines, see Section 7.7.3 [insert_select], page 293. Inserted files assume non-'preserve_block' semantics. An equivalent files setting that does preserve the ordering of lines in the file is called file_preserve_block. This was added in CFEngine Core 3.5.x.

7.7.3 insert_select (body template)

Type: (ext body)

'insert_if_startwith_from_list'

Type: slist

Allowed input range: .*

Synopsis: Insert line if it starts with a string in the list

Example:

```
body insert_select example
{
insert_if_startwith_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The list contains literal strings to search for in the secondary file (the file being read via the insert_type attribute, not the main file being edited). If a string with matching starting characters is found, then that line from the secondary file will be inserted at the present location in the primary file.

insert_if_startswith_from_list is ignored unless insert_type is file (see Section 7.7.2 [insert_type in insert_lines], page 292), or the promiser is a multi-line block.

'insert_if_not_startwith_from_list'

Type: slist

Allowed input range: .*

Synopsis: Insert line if it DOES NOT start with a string in the list

Example:

```
body insert_select example
{
insert_if_not_startwith_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The complement of `insert_if_startwith_from_list`. If the start of a line does *not* match one of the strings, that line is inserted into the file being edited.

`insert_if_not_startswith_from_list` is ignored unless `insert_type` is `file` (see Section 7.7.2 [insert_type in insert_lines], page 292), or the promiser is a multi-line block.

'`insert_if_match_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if it fully matches a regex in the list

Example:

```
body insert_select example
{
insert_if_match_from_list => { ".*find_.*_1.*", ".*find_.*_2.*" };
}
```

Notes:

The list contains literal strings to search for in the secondary file (the file being read via the `insert_type` attribute, not the main file being edited). If the regex matches a *complete* line of the file, that line from the secondary file will be inserted at the present location in the primary file. That is, the regex's in the list are anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

`insert_if_match_from_list` is ignored unless `insert_type` is `file`, see Section 7.7.2 [insert_type in insert_lines], page 292), or the promiser is a multi-line block.

'`insert_if_not_match_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if it DOES NOT fully match a regex in the list

Example:

```
body insert_select example
```

```
{
insert_if_not_match_from_list => { ".*find_.*_1.*", ".*find_.*_2.*" };
}
```

Notes:

The complement of `insert_if_match_from_list`. If the line does *not* match a line in the secondary file, it is inserted into the file being edited.

`insert_if_not_match_from_list` is ignored unless `insert_type` is `file` (see Section 7.7.2 [insert_type in insert_lines], page 292), or the promiser is a multi-line block.

'`insert_if_contains_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if a regex in the list match a line fragment

Example:

```
body insert_select example
{
insert_if_contains_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The list contains literal strings to search for in the secondary file (the file being read via the `insert_type` attribute, not the main file being edited). If the string is found in a line of the file, that line from the secondary file will be inserted at the present location in the primary file.

`insert_if_contains_from_list` is ignored unless `insert_type` is `file` (see Section 7.7.2 [insert_type in insert_lines], page 292), or the promiser is a multi-line block.

'`insert_if_not_contains_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if a regex in the list DOES NOT match a line fragment

Example:

```
body insert_select example
{
insert_if_not_contains_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The complement of `insert_if_contains_from_list`. If the line is *not* found in the secondary file, it is inserted into the file being edited.

`insert_if_not_contains_from_list` is ignored unless `insert_type` is `file` (see Section 7.7.2 [insert_type in insert_lines], page 292), or the promiser is a multi-line block.

7.7.4 `location` (body template)

Type: (ext body)

'`before_after`'

Type: (menu option)

Allowed input range:

```
before
after
```

Synopsis: Menu option, point cursor before of after matched line

Default value: after

Example:

```
body location append

{
#...
before_after => "before";
}
```

Notes:

Determines whether an edit will occur before or after the currently matched line.

'`first_last`'

Type: (menu option)

Allowed input range:

```
first
last
```

Synopsis: Menu option, choose first or last occurrence of match in file

Default value: last

Example:

```
body location example
{
first_last => "last";
}
```

Notes:

In multiple matches, decide whether the first or last occurrence of the matching pattern in the case affected by the change. In principle this could be generalized to more cases but this seems like a fragile quality to evaluate, and only these two cases are deemed of reproducible significance.

'select_line_matching'

Type: string

Allowed input range: `.*`

Synopsis: Regular expression for matching file line location

Example:

```
# Editing

body location example
{
select_line_matching => "Expression match.* whole line";
}

# Measurement promises

body match_value example
{
select_line_matching => "Expression match.* whole line";
}
```

Notes:

The expression must match a whole line, not a fragment within a line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

This attribute is mutually exclusive of `select_line_number`.

7.7.5 whitespace_policy

Type: (option list)

Allowed input range:

```
ignore_leading
ignore_trailing
ignore_embedded
exact_match
```

Synopsis: Criteria for matching and recognizing existing lines

Example:

```
bundle edit_line Insert(service, filename)
{
insert_lines:

  "$(service).* $(filename)"

      whitespace_policy => { "ignore_trailing", "ignore_embedded" };

}
```

Notes:

The white space matching policy applies only to `insert_lines`, as a convenience. It works by rewriting the insert string as a regular expression when *matching* lines (that is, when determining if the line is already in the file), but leaving the string as specified when actually inserting it.

Simply put, the 'does this line exist' test will be changed to a regexp match. The line being tested will optionally have `"\s*"` prepended or appended if `ignore_leading` or `ignore_trailing` is specified, and if `ignore_imbedded` is used then all embedded whitespaces are replaced with '`\s+`'. You may specify more than one `whitespace_policy` – they are additive.

Any regular expression meta-characters that exist in your input line will be escaped, so that you can still, for example, safely insert a line like '`"authpriv.* /var/log/something"`' into your syslog config file.

History: This attribute was introduced in CFEngine version 3.0.5 (2010)

Default value:

`exact_match` (so unless you use this new attribute, your `insert_line` promises should behave as before.

7.8 field_edits promises in 'edit_line'

Certain types of text file (e.g. the 'passwd' and 'group' files in Unix) are tabular in nature, with field separators (e.g. ':' or ','). This promise assumes a parameterizable model for editing the fields of such files, using a regular expression to separate major fields and a character to separate sub-fields. First you match the line with a regular expression. The regular expression must match the entire line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). Then a field_edits body describes the separators for fields and one level of sub-fields, along with policies for editing these fields, ordering the items within them etc.

```
field_edits:

    "regex matching line"

              edit_field => body;
```

```
bundle agent example

{
vars:

 "userset" slist => { "one-x", "two-x", "three-x" };

files:

  "/tmp/passwd"

       create    => "true",
       edit_line => SetUserParam("mark","6","/set/this/shell");

  "/tmp/group"

       create    => "true",
       edit_line => AppendUserParam("root","4","@(userset)");
}

#########################################################

bundle edit_line SetUserParam(user,field,val)
  {
  field_edits:
```

```
   "$(user):.*"

      # Set field of the file to parameter

      edit_field => col(":","$(field)","$(val)","set");
  }

########################################################

bundle edit_line AppendUserParam(user,field,allusers)
  {
  vars:

    "val" slist => { @(allusers) };

  field_edits:

   "$(user):.*"

      # Set field of the file to parameter

      edit_field => col(":","$(field)","$(val)","alphanum");

  }

########################################
# Bodies
########################################

body edit_field col(split,col,newval,method)

{
field_separator => "$(split)";
select_field    => "$(col)";
value_separator  => ",";
field_value     => "$(newval)";
field_operation => "$(method)";
extend_fields => "true";
}
```

Field editing allows us to edit tabular files in a unique way, adding and removing data from addressable fields. The 'passwd' and 'group' files are classic examples of tabular files, but there are many ways to use this feature, e.g. edit a string

```
VARIABLE="one two three"
```

View this line as a tabular line separated by ‘"’ and with sub-separator given by the space.

7.8.1 edit_field (body template)

Type: (ext body)

‘allow_blank_fields’

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false allow blank fields in a line (do not purge)

Default value: false

Example:

```
body edit_field example
{
# ...
allow_blank_fields => "true";
}
```

Notes:

When editing a file using the field or column model, blank fields, especially at the start and end are generally discarded. If this is set to true, CFEngine will retain the blank fields and print the appropriate number of field separators.

‘extend_fields’

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false add new fields at end of line if necessary to complete edit

Default value: false

Example:

```
body edit_field example
{
extend_fields => "true";
}
```

Notes:

If a user specifies a field that does not exist, because there are not so many fields, this allows the number of fields to be extended. Without this setting, CFEngine will issue an error if a non-existent field is referenced. Blank fields in a tabular file can be eliminated or kept depending in this setting. If in doubt, set this to true.

'field_operation'

Type: (menu option)

Allowed input range:

```
prepend
append
alphanum
delete
set
```

Synopsis: Menu option policy for editing subfields

Default value: none

Example:

```
body edit_field example
{
field_operation => "append";
}
```

Notes:

The method by which to edit a field in multi-field/column editing of tabular files. The methods mean:

'append' - append the specified value to the end of the field/column, separating (potentially) multiple values with 'value_separator'

'prepend' - prepend the specified value at the beginning of the field/column, separating (potentially) multiple values with 'value_separator'

'alphanum' - insert the specified value into the field/column, keeping all the values (separated by 'value_separator') in alphanumerically sorted order)

'set' - replace the entire field/column with the specified value

'delete' - delete the specified value (if present) in the specified field/column

Default value:

append

'field_separator'

Type: string

Allowed input range: .*

Synopsis: The regular expression used to separate fields in a line

Default value: none

Example:

```
body edit_field example
{
field_separator => ":";
}
```

Notes:

Most tabular files are separated by simple characters, but by allowing a general regular expression one can make creative use of this model to edit all kinds of line-based text files.

'field_value'

Type: string

Allowed input range: .*

Synopsis: Set field value to a fixed value

Example:

```
body edit_field example(s)
{
field_value => "$(s)";
```

```
}
```

Notes:

Set a field to a constant value, e.g. reset the value to a constant default, empty the field, or set it fixed list.

'select_field'

Type: int

Allowed input range: 0,99999999

Synopsis: Integer index of the field required 0..n (default starts from 1)

Example:

```
body field_edits example
{
select_field => "5";
}
```

Notes:

Numbering starts from 1 (not from 0).

'start_fields_from_zero'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: If set, the default field numbering starts from 0

Example:

```
body edit_field col(split,col,newval,method)

{
field_separator    => "$(split)";
select_field       => "$(col)";
```

```
value_separator    => ",";
field_value        => "$(newval)";
field_operation    => "$(method)";
extend_fields      => "true";
allow_blank_fields => "true";
start_fields_from_zero => "true";
}
```

Notes:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

The numbering of fields is a matter for consistency and convention. Arrays are usually thought to start with first index equal to zero (0), but the first column in a file would normally be 1. By setting this option, you can tell CFEngine that the first column should be understood as number 0 instead, for consistency with other array functions.

'value_separator'

Type: string

Allowed input range: `^.$`

Synopsis: Character separator for subfields inside the selected field

Default value: none

Example:

```
body field_edit example
{
value_separator => ",";
}
```

Notes:

For example, elements in the group file are separated by ':', but the lists of users in these fields are separated by ','.

7.9 replace_patterns promises in 'edit_line'

This promise refers to arbitrary text patterns in a file. The pattern is expressed as a PCRE regular expression.

```
replace_patterns:

 "search pattern"

    replace_with => replace_body,
    ...;
```

```
bundle edit_line upgrade_cfexecd
{
  replace_patterns:

    "cfexecd" replace_with => value("cf-execd");
}

########################################

body replace_with value(x) # defined in cfengine_stdlib.cf
{
replace_value => "$(x)";
occurrences => "all";
}
```

This is a straightforward search and replace function. Only the portion of the line that matches the pattern in the promise will be replaced - the remainder of the line will not be affected. You can also use PCRE lookbehind and lookahead patterns to restrict the lines upon which the pattern will match.

NOTE: In `replace_patterns` promises, the regular expression may match a line fragment, that is, it is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

7.9.1 `replace_with` (body template)

Type: (ext body)

'`occurrences`'

Type: (menu option)

Allowed input range:

```
all
first
```

Synopsis: Menu option to replace all occurrences or just first (NB the latter is non-convergent)

Default value: all

Example:

```
body replace_with example
{
occurrences => "first";             # Warning! Using "first" is non-convergent
}
```

Notes:

A policy for string replacement.

Default value:

The default value is "all". Using "first" is generally unwise, as it is possibly non-convergent (it will change a different matching string each time the promise is executed, and may not "catch up" with whatever external action is altering the text the promise applies to).

'replace_value'

Type: string

Allowed input range: `.*`

Synopsis: Value used to replace regular expression matches in search

Example:

```
body replace_with example(s)
{
replace_value => "$(s)";
}
```

Notes:

7.10 Miscelleneous in edit_xml promises

The use of XML documents in systems configuration is widespread. XML documents represent data that is complex and can be structured in various ways. The XML based editing offers a powerful environment for editing hierarchical and structured XML datasets.

7.10.1 build_xpath

Type: string

Allowed input range: (arbitrary string)

Synopsis: Build an XPath within the XML file

Example:

```
body build_xpath example(s)
  {
  build_xpath => "$(s)";
  }
```

Notes:

Please note that when `build_xpath` is defined as an attribute, within an `edit_xml` promise body, the tree described by the specified XPath will be verified and built BEFORE other `edit_xml` promises within same promise body. Therefore, the file will not be empty during the execution of such promises.

7.10.2 select_xpath

Type: string

Allowed input range: (arbitrary string)

Synopsis: Select the XPath node in the XML file to edit

Example:

```
body select_xpath example(s)
{
select_xpath => "$(s)";
}
```

Notes:

Edits to the XML document take place within the selected node. Please note that this attribute is not used when inserting XML content into an empty file.

7.11 build_xpath promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that a balanced XML tree, described by the given XPath, will be present within the XML file. If the document is empty, the default promise is to build the XML tree within the document. If the document is not empty, the default promise is to verify the given XPath, and if necessary, locate an insertion node and build the necessary portion of xlm tree within selected node. The insertion node is selected as the last unique node that is described by the XPath and also found within the document. The promise object referred to is a literal string representation of an XPath.

```
bundle edit_xml example
  {
  build_xpath:
    "/Server/Service/Engine/Host[ @name=\"cfe_host\" | Alias = cfe_alias ]";
  }
```

Note that typically, only a single XPath is built in each `build_xpath` promise (you may of course have multiple promises that each build an XPath). Also, please note that the supported syntax used in building an XPath is currently limited to a simple and compact format, as shown in the above example. The XPath must begin with '/', as it is verified and built using an absolute path, from the root node of the document.

7.12 `delete_tree` promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that a balanced XML tree, containing the matching subtree, will not be present in the specified node within the XML file. If the subtree is found, the default promise is to remove the containing tree from within the specified node. The specified node is determined by body-attributes. The promise object referred to is a literal string representation of a balanced XML subtree.

```
bundle edit_xml example
  {
  delete_tree:
    "<Host name=\"cfe_host\"></Host>"

    select_xpath => "/Server/Service/Engine";
  }
```

Note that typically, only a single tree, within a single specified node, is deleted in each `delete_tree` promise (you may of course have multiple promises that each delete a tree).

7.13 `insert_tree` promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that a balanced XML tree, containing the matching subtree, will be present in the specified node within the XML file. If the subtree is not found, the default promise is to insert the tree into the specified node. The specified node is determined by

body-attributes. The promise object referred to is a literal string representation of a balanced XML tree.

```
bundle edit_xml example
  {
  insert_tree:
    "<Host name=\"cfe_host\"><Alias>cfe_alias</Alias></Host>"

    select_xpath => "/Server/Service/Engine";
  }
```

Note that typically, only a single tree, within a single specified node, is inserted in each `insert_tree` promise (you may of course have multiple promises that each insert a tree).

7.14 `delete_attribute` promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that an attribute, with the given name, will not be present in the specified node within the XML file. If the attribute is found, the default promise is to remove the attribute, from within the specified node. The specified node is determined by body-attributes. The promise object referred to is a literal string representation of the name of the attribute to be deleted.

```
bundle edit_xml example
  {
  delete_attribute:
    "name"

    select_xpath => "/Server/Service/Engine/Host";
  }
```

Note that typically, only a single attribute, within a single specified node, is deleted in each `delete_attribute` promise (you may of course have multiple promises that each delete an attribute).

7.15 `set_attribute` promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that an attribute, with the given name and value, will be present in the specified node within the XML file. If the attribute is not found, the default promise is to insert the attribute, into the specified node. If the attribute is already present, the default promise is to insure that the attribute value is set to the given value. The specified node and attribute value are each determined by body-attributes. The promise object referred to is a literal string representation of the name of the attribute to be set.

```
bundle edit_xml example
  {
  set_attribute:
    "name"

    attribute_value => "cfe_host",
    select_xpath => "/Server/Service/Engine/Host";
  }
```

Note that typically, only a single attribute, within a single selected node is set in each `set_attribute` promise (you may of course have multiple promises that each set an attribute).

7.15.1 attribute_value

Type: string

Allowed input range: (arbitrary string)

Synopsis: Value of the attribute to be inserted into the XPath node of the XML file

Example:

```
body attribute_value example(s)
{
attribute_value => "$(s)";
}
```

Notes:

7.16 delete_text promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that a value string, containing the matching substring, will not be present in the specified node within the XML file. If the substring is found, the default promise is to remove the existing value string, from within the specified node. The

specified node is determined by body-attributes. The promise object referred to is a literal string of text.

```
bundle edit_xml example
  {
  delete_text:
    "text content to match, as a substring, of text to be deleted from specified node"

    select_xpath => "/Server/Service/Engine/Host/Alias";
  }
```

Note that typically, only a single value string, within a single specified node, is deleted in each `delete_text` promise (you may of course have multiple promises that each delete a value string).

7.17 `set_text` promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that a matching value string will be present in the specified node within the XML file. If the existing value string does not exactly match, the default promise is to replace the existing value string, within the specified node. The specified node is determined by body-attributes. The promise object referred to is a literal string of text.

```
bundle edit_xml example
  {
  set_text:
    "text content to replace existing text, including whitespace, within selected node"

    select_xpath => "/Server/Service/Engine/Host/Alias";
  }
```

Note that typically, only a single value string, within a single selected node, is set in each `set_text` promise (you may of course have multiple promises that each set a value string).

7.18 `insert_text` promises in 'edit_xml'

This promise is part of the XML-editing model. It assures that a value string, containing the matching substring, will be present in the specified node within the XML file. If the substring is not found, the default promise is to append the substring at the end of the existing value string, within the specified node. The specified node is determined by body-attributes. The promise object referred to is a literal string of text.

```
bundle edit_xml example
  {
  insert_text:
    "text content to be appended to existing text, including whitespace, within specified node"

    select_xpath => "/Server/Service/Engine/Host/Alias";
  }
```

Note that typically, only a single value string, within a single specified node, is inserted in each `insert_text` promise (you may of course have multiple promises that each insert a value string).

7.19 `interfaces` promises in 'agent'

Interfaces promises describe the configurable aspects relating to network interfaces. Most workstations and servers have only a single network interface, but routers and multi-homed hosts often have multiple interfaces. Interface promises include attributes such as the IP address identity, assumed netmask and routing policy in the case of multi-homed hosts. For virtual machines and hosts, the list of interfaces can be quite large.

```
interfaces:

  "interface name"

    tcp_ip => tcp_ip_body,
    ...;
```

For future use.

For future use.

7.19.1 `tcp_ip` (body template)

Type: (ext body)

'ipv4_address'

Type: string

Allowed input range: [0-9.]+/[0-4]+

Synopsis: IPv4 address for the interface

Example:

```
body tcp_ip example
{
ipv4_address => "123.456.789.001";
}
```

Notes:

The address will be checked and if necessary set. Today few hosts will be managed in this way: address management will be handled by other services like DHCP.

'ipv4_netmask'

Type: string

Allowed input range: [0-9.]+/[0-4]+

Synopsis: Netmask for the interface

Example:

```
body tcp_ip example
{
ipv4_netmask => "255.255.254.0";
}
```

Notes:

In many cases the CIDR form of address will show the netmask as '/23', but this offers and 'old style' alternative.

'ipv6_address'

Type: string

Allowed input range: [0-9a-fA-F:]+/[0-9]+

Synopsis: IPv6 address for the interface

Example:

```
"eth0"

   ipv6_address => "2001:700:700:3:211:63ff:feeb:5d18/64";
```

Notes:

7.20 methods promises in 'agent'

Methods are compound promises that refer to whole bundles of promises. Methods may be parameterized. Methods promises are written in a form that is ready for future development. The promiser object is an abstract identifier that refers to a collection (or pattern) of lower level objects that are affected by the promise-bundle. Since the use of these identifiers is for the future, you can simply use any string here for the time being.

```
methods:

  "any"

     usebundle => method_id("parameter",...);
```

Methods are useful for encapsulating repeatedly used configuration issues and iterating over parameters.

In CFEngine 2 methods referred to separate sub-programs executed as separate processes. Methods are now implemented as bundles that are run inline.

```
bundle agent example
{
vars:

 "userlist" slist => { "mark", "jeang", "jonhenrik", "thomas", "eben" };

methods:

 "any" usebundle => subtest("$(userlist)");

}

###########################################
```

```
bundle agent subtest(user)

{
commands:

 "/bin/echo Fix $(user)";

reports:

 linux::

  "Finished doing stuff for $(user)";
}
```

Methods offer powerful ways to encapsulate multiple issues pertaining to a set of parameters.

Because a method is just an encapsulation, there is a subtlety about how to interpret a successful method invocation. Before version 3.1.0, a method was considered repaired if executed (like `commands`), however this led to unnecessary logging of executions, even if not actual encapsulated promise was kept. In version 3.1.0 this has been changed so that a method promise is considered kept if the method is expanded. A method promise is thus never considered repaired.

Starting from version 3.1.0, methods may be specified using variables. Care should be exercised when using this approach. In order to make the function call uniquely classified, CFEngine requires the promiser to contain the variable name of the method if the variable is a list.

```
bundle agent default
{
vars:
    "m" slist  => { "x", "y" };
    "p" string => "myfunction";

methods:
    "set of $(m)" usebundle => $(m) ("one");
    "any"         usebundle => $(p)("two");

}
```

7.20.1 inherit

Type: (menu option)

Allowed input range:

```
true
false
yes
no
```

```
on
off
```

Synopsis: If true this causes the sub-bundle to inherit the private classes of its parent

Example:

```
bundle agent name
{
methods:

  "group name" usebundle => my_method,
                 inherit => "true";
}


body edit_defaults example
{
inherit => "true";
}
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

Default value: false

The `inherit` constraint can be added to the CFEngine code in two places: for `edit_defaults` and in `methods` promises. If set to true, it causes the child-bundle named in the promise to inherit (only) the classes of the parent bundle. Inheriting the variables is unnecessary as the child can always access the parent's variables by a qualified reference using its bundle name, e.g. '`$(bundle.variable)`'.

7.20.2 usebundle

Type: (ext bundle) (Separate Bundle)

7.20.3 useresult

Type: string

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: Specify the name of a local variable to contain any result/return value from the child

Example:

```
body common control
{
bundlesequence => { "test" };
}
```

```
bundle agent test
{
methods:

   "any" usebundle => child,
    useresult => "my_return_var";


reports:

  cfengine_3::

    "My return was: \"$(my_return_var[1])\" and \"$(my_return_var[2])\"";

}

bundle agent child
{
reports:

 cfengine_3::

   # Map these indices into the useresult namespace

   "this is a return value"
      bundle_return_value_index => "1";

   "this is another return value"
      bundle_return_value_index => "2";

}
```

Notes:

History: Was introduced in 3.4.0 (2012)

It is currently believed that only scalar return values make sense, hence return values are limited to scalars.

7.21 `outputs` promises in '`agent`'

Outputs promises allow promises to make meta-promises about their output levels. More simply, you can switch on '`verbose`' or '`inform`' level output to named promises, or whole bundles for debugging purposes.

If you use the '-I' or '-v' command line options, then CFEngine will generate informative or verbose output for all the promises it is processing. This can be a daunting collection of data when dealing with even a medium-sized set of promises.

Output promises enable you to selectively debug individually named promises (or bundles), thus eliminating the need for scanning unrelated CFEngine output.

```
outputs:

  "run_agent";             # Promise handle, verbose (default) output

  "web_server"             # Bundle handle, inform output
     output_level => "inform",
     promiser_type => "bundle";
```

A very handy paradigm is to include outputs promises in every bundle, and guard them with classes. For example:

```
bundle agent some_function
{
vars:
    ...
classes:
    ...
outputs:
  debug_some_function::
      "some_function"
          output_level => "verbose",
          promiser_type => "bundle";
files:
    ...
}
```

You can then execute your promises normally with no extra output, but should you wish to temporarily enable debugging, you can simply do so from the command line by specifying '-D debug_some_function'. You can also supply multiple arguments to '-D' to debug multiple bundles. Of course, you can also provide much finer-grained control by creating outputs promises on specific promise handles.

The default behaviour is to print verbose output for listed promise handles, See Section 6.6.5 [handle in *], page 188, for bundle names.

History This was introduced in Nova version 1.1.3 (2010), Community version 3.4.0 (2012)

7.21.1 output_level

Type: (menu option)

Allowed input range:

```
verbose
debug
inform
```

Default value: verbose

Synopsis: Output level to observe for the named promise or bundle (meta-promise)

Example:

```
commands:

  "/etc/init.d/agent start"

    handle => "run_agent",
    ifvarclass => "need_to_run_agent";

outputs:

  "run_agent"

    output_level => "inform";
```

Notes:

With no attribute, `verbose` output is assumed.

7.21.2 promiser_type

Type: (menu option)

Allowed input range:

```
promise
bundle
```

Default value: promise

Synopsis: Output level to observe for the named promise or bundle (meta-promise)

Example:

```
outputs:
```

```
  "web_server"

     promiser_type => "bundle";
```

Notes:

Without this attribute, CFEngine assumes a list of promises to report on (because there may be a promise for a thing that has the same name as a bundle, you must explicitly specify when you want to report on a bundle of promises).

7.22 packages promises in 'agent'

```
vars:

 "match_package" slist => {
                          "apache2",
                          "apache2-mod_php5",
                          "apache2-prefork",
                          "php5"
                          };
packages:

   "$(match_package)"

        package_policy => "add",
        package_method => yum;
```

Software packaging is a core paradigm in operating system release management today, and CFEngine supports a generic approach to integration with native operating support for packaging. Package promises allow CFEngine to make promises the state of software packages *conditionally*, given the assumption that a native package manager will perform the actual manipulations. Since no agent can make unconditional promises about another, this is the best that can be achieved.

Packages are treated as black-boxes with three labels:

- A package name.
- A version string.
- An architecture name.

Package managers are treated as black boxes that may support some or all of the following promise types:

- List installed packages
- Add packages

- Delete packages
- Reinstall (repair) packages
- Update packages
- Patch packages
- Verify packages

If these services are promised by a package manager, `cf-agent` promises to use the service and encapsulate it within the overall CFEngine framework. It is possible to set classes based on the return code of a package-manager command in a very flexible way. See the `kept_returncodes`, `repaired_returncodes` and `failed_returncodes` attributes.

Domain knowledge

CFEngine does not maintain operating system specific expert knowledge internally, rather it uses a generic model for dealing with promises about packages (which depend on the behaviour of an external package manager). The approach is to define package system details in body-constraints that can be written once and for all, for each package system.

Package promises are like `commands` promises in the sense that CFEngine promises nothing about the outcome of executing a command. All it can promise is to interface with it, starting it and using the results in good faith. Packages are basically 'outsourced', to invoke IT parlance.

The possibility of a CFEngine package format that enables more guaranteeable behaviour for special purposes has not been excluded for the future, but in any case `cf-agent` must support native package formats used by operating system maintainers as these are a core part of modern operating systems.

Behaviour

A package promise consists of a name, a version and an architecture, *(n,v,a)*, and behaviour to be promised about packages that match criteria based on these. The components *(n,v,a)* can be determined in one of two different ways:

- They may be specified independently, e.g.

```
packages:

  "mypackage"

     package_policy => "add",
     package_method => rpm,
     package_select => ">=",
     package_architectures => { "x86_64", "i586" },
     package_version => "1.2.3";
```

- They may be extracted from a package identifier (promiser) or filename, using pattern matching. For example, a promiser "7-Zip-4.50-x86_64.msi" and a package_method containing the following.

```
package_name_regex => "^(\S+)-(\d+\.?)+";
package_version_regex => "^\S+-((\d+\.?)+)";
package_arch_regex => "^\S+-[\d\.]+-(.*).msi";
```

When scanning a list of installed packages different managers present the information *(n,v,a)* in quite different forms and pattern extraction is necessary. When making a promise about a specific package, the CFEngine user may choose one or the other model.

Smart and dumb package systems

Package managers vary enormously in their capabilities and in the kinds of promises they make. There are broadly two types

- Smart package systems tha resolve dependencies and require only a symbolic package name.
- Dumb package managers that do not resolve dependencies and need filename input.

Normal ordering for packages is the following:

- Delete
- Add
- Update
- Patch

Promise repair logic

We can discuss package promise repair in the following table.

Identified package matches version constraints

add	never
delete	=,>=,<=
reinstall	=,>=,<=
upgrade	=,>=,<=
patch	=,>=,<=

Identified package matched by name, but not version

Command	Dumb manager	Smart manager
add	unable	Never
delete	unable	Attempt deletion
reinstall	unable	Attempt delete/add
upgrade	unable	Upgrade if capable
patch	unable	Patch if capable

Package not installed

Command	Dumb manager	Smart manager
add	Attempt to install named	Install any version
delete	unable	unable
reinstall	Attempt to install named	unable
upgrade	unable	unable
patch	unable	unable

```
bundle agent packages
{
vars:

 # Test the simplest case -- leave everything to the yum smart manager

 "match_package" slist => {
                          "apache2",
```

```
                              "apache2-mod_php5",
                              "apache2-prefork",
                              "php5"
                              };
packages:

  "$(match_package)"

     package_policy => "add",
     package_method => yum;

}
```

Packages promises can be very simple if the package manager is of the smart variety that handles details for you. If you need to specify architecture and version numbers of packages, this adds some complexity, but the options are flexible and designed for maximal adaptability.

Patching

Some package systems also support the idea of 'patches'. These might be formally different objects to packages. A patch might contain material for several packages and be numbered differently. When you select patching-policy the package name (promiser) can be a regular expression that will match possible patch names, otherwise identifying specific patches can be cumbersome.

Note that patching is a subtle business. There is no simple way using the patch settings to install 'all new system patches'. Here's why:

If we specify the name of a patch, then CFEngine will try to see if it exists and/or is installed. If it exists in the pending list, it will be installed. If it exists in the installed list it will not be installed. Now consider the pattern '`.*`'. This will match any installed package, so CFEngine will assume the relevant patch has been installed already. On the other hand, the pattern '`no match`' will not match an installed patch, but it will not match a named patch either.

Some systems provide a command to do this, which can be specified without specific patch arguments. If so, that command can be called periodically under `commands`. The main purposes of patching body items are:

- To install specific named patches in a controlled manner.
- To generate reports of available and installed patches during system reporting.

Installers without package/patch arguments

CFEngine supports the syntax '`$`' at the end of a command to mean that no package name arguments should be used or appended after the dollar. This is because some commands require a list of packages, while others require an empty list. The default behaviour is to try to append the name of one or more packages to the command, depending on whether the policy is for individual or bulk installation.

Default package method

As of core 3.3.0, if no `package_method` is defined, CFEngine will look for a method called '`generic`'. Such a method is defined in the standard library for supported operating systems.

7.22.1 package_architectures

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Select the architecture for package selection

Example:

```
packages:

  "$(exact_package)"

     package_policy => "add",
     package_method => rpm,
     package_architectures => { "x86_64" };
```

Notes:

It is possible to specify a list of packages of different architectures if it is desirable to install multiple architectures on the host. If no value is specified, CFEngine makes no promise about the result; the package manager's behaviour prevails.

7.22.2 package_method (body template)

Type: (ext body)

'package_add_command'

Type: string

Allowed input range: .+

Synopsis: Command to install a package to the system

Example:

```
body package_method rpm
{
package_add_command => "/bin/rpm -i ";
}
```

Notes:

This command should install a package when appended with the package reference id, formed using the package_name_convention, using the model of (name,version,architecture). If package_file_repositories is specified, the package reference id will include the full path to a repoistory containing the package.

Package managers generally expect the name of a package to be passed as a parameter. However, in some cases we do not need to pass the name of a particular package to the command. Ending the command string with '$' prevents CFEngine from appending the package name to the string.

'package_arch_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package architecture string

Example:

```
body package_method rpm

{
package_list_arch_regex    => "[^.]+\.([^.]+)";
}
```

Notes:

This is for use when extracting architecture from the name of the promiser, i.e. when the architecture is not specified using the `package_architectures` list. It is a regular expression that contains exactly one parenthesized back reference which marks the location in the *promiser* at which the architecture is specified. The regex may match a portion of the string (i.e., it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). If no architecture is specified for the given package manager, then do not define this.

'package_changes'

Type: (menu option)

Allowed input range:

```
individual
bulk
```

Synopsis: Menu option - whether to group packages into a single aggregate command

Example:

```
body package_method rpm

{
package_changes => "bulk";
```

```
}
```

Notes:

This indicate whether the package manager is capable of handling package operations in bulk, i.e. with by given multiple arguments. If this is set to 'bulk' then multiple arguments will be passed to the package commands. If set to 'individual' packages will be handled one by one. This might add a significant overhead to the operations, and also affect the ability of the operating system's package manager to handle dependencies.

'package_delete_command'

Type: string

Allowed input range: .+

Synopsis: Command to remove a package from the system

Example:

```
body package_method rpm

{
package_delete_command => "/bin/rpm -e --nodeps";
}
```

Notes:

The command that deletes a package from the system when appended with the package reference identifier specified by package_name_convention.

Package managers generally expect the name of a package to be passed as a parameter. However, in some cases we do not need to pass the name of a particular package to the command. Ending the command string with '$' prevents CFEngine from appending the package name to the string.

'package_delete_convention'

Type: string

Allowed input range: (arbitrary string)

Synopsis: This is how the package manager expects the package to be referred to in the deletion part of a package update, e.g. $(name)

Example:

```
body package_method freebsd

{
```

```
package_file_repositories => { "/path/to/packages" };
package_name_convention => "$(name)-$(version).tbz";
package_delete_convention => "$(name)-$(version)";
}
```

Notes:

This attribute is used when `package_policy` is '`delete`', or `package_policy` is '`update`' and `package_file_repositories` is set and `package_update_command` is not set. It is then used to set the pattern for naming the package in the way expected by the package manager during the deletion of existing packages.

Three special variables are defined from the extracted data, in a private context for use: '`$(name)`', '`$(version)`' and '`$(arch)`'. '`version`' and '`arch`' is the version and arch (if `package_list_arch_regex` is given) of the already installed package. Additionally, if `package_file_repositories` is defined, '`$(firstrepo)`' can be prepended to expand the first repository containing the package, e.g. '`$(firstrepo)$(name)-$(version)-$(arch).msi`'.

If this is not defined, it defaults to the value of `package_name_convention`.

'`package_file_repositories`'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of machine-local directories to search for packages

Example:

```
body package_method filebased
{
package_file_repositories => { "/package/repos1", "/packages/repos2" };
}
```

Notes:

If specified, CFEngine will assume that the package installation occurs by filename and will search the named paths for a package matching the pattern `package_name_convention`. If found the name will be prefixed to the package name in the package commands.

'`package_installed_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression which matches packages that are already installed

Example:

```
body package_method yum
{
package_installed_regex => ".*installed.*";
}
```

Notes:

This regular expression must match complete lines in the output of the list command that are actually installed packages (that is, the regex is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). If all the lines match then the regex can be set of '.*', however most package systems output prefix lines and a variety of human padding that needs to be ignored.

'package_default_arch_command'

Type: string

Allowed input range: "?(/.*)

Synopsis: Command to detect the default packages' architecture

Example:

```
body package_method dpkg
{
  package_default_arch_command => "/usr/bin/dpkg --print-architecture";

  # ...
}
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0.0 (2012)

This command allows CFEngine to detect default architecture of packages managed by package manager. As an example, multiarch-enabled dpkg only lists architectures explicitly for multiarch-enabled packages.

In case this command is not provided, CFEngine treats all packages without explicit architecture set as belonging to implicit "default" architecture.

'package_list_arch_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package architecture string

Example:

```
body package_method rpm
{
package_list_arch_regex    => "[^|]+\|[^|]+\|[^|]+\|[^|]+\|\s+([^\s]+).*";
}
```

Notes:

A regular expression that contains exactly one parenthesized back reference which marks the location in the listed package at which the architecture is specified. The regular expression may match a portion of the string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). If no architecture is specified for the given package manager, then do not define this regex.

'package_list_command'

Type: string

Allowed input range: `.+`

Synopsis: Command to obtain a list of available packages

Example:

```
body package_method rpm

{
package_list_command => "/bin/rpm -qa --queryformat \"%{name} %{version}-%{release}\n\"";
}
```

Notes:

This command should provide a complete list of the packages installed on the system. It might also list packages that are not installed. Those should be filtered out using the `package_installed_regex`.

Package managers generally expect the name of a package to be passed as a parameter. However, in some cases we do not need to pass the name of a particular package to the command. Ending the command string with '$' prevents CFEngine from appending the package name to the string.

'package_list_name_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package name string

Example:

```
body package_method rpm

{
package_list_name_regex    => "([^\s]+).*";
}
```

Notes:

A regular expression that contains exactly one parenthesized back reference which marks the name of the package from the package listing. The regular expression may match a portion of the string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'package_list_update_command'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Command to update the list of available packages (if any)

Example:

```
body package_method xyz
{
debian|ubuntu::

package_list_update_command => "/usr/bin/apt-get update";
package_list_update_ifelapsed => "240";             # 4 hours
}
```

Notes:

Not all package managers update their list information from source automatically. This command allows a separate update command to be executed at intervals determined by package_list_update_ifelapsed.

'package_list_update_ifelapsed'

Type: int

Allowed input range: -99999999999,9999999999

Synopsis: The ifelapsed locking time in between updates of the package list

Example:

```
body package_method xyz
{
debian|ubuntu::

package_list_update_command => "/usr/bin/apt-get update";
package_list_update_ifelapsed => "240";             # 4 hours
}
```

Notes:

Not all package managers update their list information from source automatically. This command allows a separate update command to be executed at intervals determined by `package_list_update_ifelapsed`.

'`package_list_version_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package version string

Example:

```
body package_method rpm

{
package_list_version_regex => "[^\s]+ ([^.]+).*";
}
```

Notes:

This regular expression should containe exactly one parenthesized back-reference that marks the version string of packages listed as installed. The regular expression may match a portion of the string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39)

'`package_name_convention`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: This is how the package manager expects the package to be referred to, e.g. $(name).$(arch)

Example:

```
body package_method rpm

{
package_name_convention => "$(name).$(arch).rpm";
}
```

Notes:

This sets the pattern for naming the package in the way expected by the package manager. Three special variables are defined from the extracted data, in a private context for use: '$(name)', '$(version)' and '$(arch)'. Additionally, if package_file_repositories is defined, '$(firstrepo)' can be prepended to expand the first repository containing the package, e.g. '$(firstrepo)$(name)-$(version)-$(arch).msi'.

When package_policy is 'update', and package_file_repositories is specified, package_delete_convention may be used to specify a different convention for the delete command.

If this is not defined, it defaults to the value '$(name)'.

'package_name_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package name string

Example:

```
body package_method rpm
{
package_name_regex => "([^\s]).*";
}
```

Notes:

This regular expression is only used when the *promiser* contains not only the name of the package, but its version and archiecture also. In that case, this expression should contain a single parenthesized back-reference to extract the name of the package from the string. The regex may match a portion of the string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39)

'package_noverify_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression to match verification failure output

Example:

```
body package_method xyz

{
package_noverify_regex => "Package .* is not installed.*";
package_verify_command => "/usr/bin/dpkg -s";
}
```

Notes:

A regular expression to match output from a package verification command. If the output string matches this expression, the package is deemed broken. The regex must match the entire line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39)

'package_noverify_returncode'

Type: int

Allowed input range: -99999999999,9999999999

Synopsis: Integer return code indicating package verification failure

Example:

```
body package_method xyz
{
package_noverify_returncode => "-1";
package_verify_command => "/bin/rpm -V";
}
```

Notes:

For use if a package verification command uses the return code as the signal for a failed package verification.

'package_patch_arch_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract update architecture string

Example:

```
body package_method zypper
```

```
{
package_patch_arch_regex => "";
}
```

Notes:

A few package managers keep a separate notion of patches, as opposed to package updates. OpenSuSE, for example, is one of these. This provide an analogous command struct to the packages for patch updates. The regular expression must match the entire line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'package_patch_command'

Type: string

Allowed input range: .+

Synopsis: Command to update to the latest patch release of an installed package

Example:

```
body package_method zypper

{
package_patch_command => "/usr/bin/zypper -non-interactive patch";
}
```

Notes:

If the package manager supports patching, this command should patch a named package. If only patching of all packages is supported then consider running that as a batch operation in commands. Alternatively one can end the command string with a '$' symbol, which CFEngine will interpret as an instruction to not append package names.

Package managers generally expect the name of a package to be passed as a parameter. However, in some cases we do not need to pass the name of a particular package to the command. Ending the command string with '$' prevents CFEngine from appending the package name to the string.

'package_patch_installed_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression which matches packages that are already installed

Example:

```
body package_method zypper
```

```
{
package_patch_installed_regex => ".*(Installed|Not Applicable).*";
}
```

Notes:

A few package managers keep a separate notion of patches, as opposed to package updates. OpenSuSE, for example, is one of these. This provide an analogous command struct to the packages for patch updates. The regular expression must match the entire string (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'package_patch_list_command'

Type: string

Allowed input range: `.+`

Synopsis: Command to obtain a list of available patches or updates

Example:

```
 package_patch_list_command => "/usr/bin/zypper patches";
```

Notes:

This command, if it exists at all, is presumed to generate a list of available patches in a format analogous to (but not necessarily the same as) the package-list command, of patches that are available on the system. Patches might formally be available in the packagae manager's view, but if they have already been installed, CFEngine will ignore them.

Package managers generally expect the name of a package to be passed as a parameter. However, in some cases we do not need to pass the name of a particular package to the command. Ending the command string with '$' prevents CFEngine from appending the package name to the string.

'package_patch_name_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract update name string

Example:

```
body package_method zypper
```

```
{
package_patch_name_regex    => "[^|]+\|\s+([^\s]+).*";
}
```

Notes:

A few package managers keep a separate notion of patches, as opposed to package updates. OpenSuSE, for example, is one of these. This provide an analogous command struct to the packages for patch updates. The regular expression may match a partial string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'package_patch_version_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract update version string

Example:

```
body package_method zypper
{
package_patch_version_regex => "[^|]+\|[^|]+\|\s+([^\s]+).*";
}
```

Notes:

A few package managers keep a separate notion of patches, as opposed to package updates. OpenSuSE, for example, is one of these. This provide an analogous command struct to the packages for patch updates. The regular expression may match a partial string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'package_update_command'

Type: string

Allowed input range: `.+`

Synopsis: Command to update to the latest version a currently installed package

Example:

```
body package_method zypper
{
package_update_command => "/usr/bin/zypper -non-interactive update";
```

```
}
```

Notes:

If supported this should be a command that updates the version of a single currently installed package. If only bulk updates are supported, consider running this as a single command under `commands`. The package reference id is appended, with the pattern of `package_name_convention`.

When `package_file_repositories` is specified, the package reference id will include the full path to a repoistory containing the package. If `package_policy` is '`update`', and this command is not specified, the `package_delete_command` and `package_add_command` will be executed to carry out the update.

'`package_verify_command`'

Type: string

Allowed input range: `.+`

Synopsis: Command to verify the correctness of an installed package

Example:

```
body package_method rpm

{
package_verify_command => "/bin/rpm -V";
package_noverify_returncode => "-1";
}
```

Notes:

If available, this is a command to verify an already installed package. It is required only when `package_policy` is '`verify`'.

The outcome of the command is compared with `package_noverify_returncode` or `package_noverify_regex`, one of which has to be set when using this command. If the package is not installed, the command will not be run — the promise gets flagged as not kept before the verify command executes.

In order for the promise to be considered kept, the package must be installed, and the verify command must be successful according to `package_noverify_returncode` xor `package_noverify_regex`.

Package managers generally expect the name of a package to be passed as a parameter. However, in some cases we do not need to pass the name of a particular package to the command. Ending the command string with '`$`' prevents CFEngine from appending the package name to the string.

'package_version_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package version string

Example:

```
body package_method rpm
{
package_version_regex => "[^\s]+ ([^.]+).*";
}
```

Notes:

If the version of a package is not specified separately using package_version, then this should be a regular expression that contains exactly one parenthesized back-reference that matches the version string in the promiser. The regular expression may match a partial string (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'package_multiline_start'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression which matches the start of a new package in multiline output

Example:

```
body package_method solaris (pkgname, spoolfile, adminfile)
{
package_changes => "individual";
package_list_command => "/usr/bin/pkginfo -l";
package_multiline_start    =>  "\s*PKGINST:\s+[^\s]+";
...
}
```

Notes:

This pattern is used in determining when a new package record begins. It is used when package managers (like the solaris package manager) use multi-line output formats. This pattern matches the first line of a new record.

'package_commands_useshell'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Whether to use shell for commands in this body

Default value: true

Example:

History: Was introduced in 3.4.0b1.70bd7ea, Nova 2.3.0.a1.3167b00 (2012)

```
Fill me in (./bodyparts/package_commands_useshell_example.texinfo)
""
```

Notes:

History: Was introduced in 3.4.0b1.70bd7ea, Nova 2.3.0.a1.3167b00 (2012)

```
Fill me in (./bodyparts/package_commands_useshell_notes.texinfo)
""
```

'package_version_less_command'

Type: string

Allowed input range: `.+`

Synopsis: Command to check whether first supplied package version is less than second one

Example:

```
body package_method deb
{
...
package_version_less_command => "dpkg --compare-versions ${v1} lt ${v2}";
}
```

Notes:

History: Was introduced in 3.4.0 (2012)

This attribute allows to override built-in CFEngine algorithm for version comparison by calling an external command to check whether the first passed version is less than another.

Built-in algorithm does a good approximation of version comparison, but different packaging systems differ in corner cases (e.g Debian treats symbol ~ less than any other symbol and even less than empty string), so some sort of override is necessary.

Variables v1 and v2 are substituted with the first and second version to be compared. Command should return code 0 if v1 is less than v2 and non-zero otherwise.

Note that if package_version_equal_command is not specified, but package_version_less_command is, then equality will be tested by issuing less comparison twice (v1 equals to v2 if v1 is not less than v2, and v2 is not less than v1).

'package_version_equal_command'

Type: string

Allowed input range: .+

Synopsis: Command to check whether first supplied package version is equal to second one

Example:

```
body package_method deb
{
...
package_version_equal_command => "dpkg --compare-versions ${v1} eq ${v2}";
}
```

Notes:

History: Was introduced in 3.4.0 (2012)

This attribute allows to override built-in CFEngine algorithm for version comparison by calling an external command to check whether the passed versions are the same. Some package managers consider textually different versions to be the same (e.g. optional epoch component, so 0:1.0-1 and 1.0-1 versions are the same), and rules for comparing vary from package manager to package manager, so override is necessary.

Variables v1 and v2 are substituted with the versions to be compared. Command should return code 0 if versions are equal and non-zero otherwise.

Note that if package_version_equal_command is not specified, but package_version_less_command is, then equality will be tested by issuing less comparison twice (v1 equals to v2 if v1 is not less than v2, and v2 is not less than v1).

7.22.3 package_policy

Type: (menu option)

Allowed input range:

```
add
delete
reinstall
update
addupdate
```

```
               patch
               verify
```

Default value: verify

Synopsis: Criteria for package installation/upgrade on the current system

Example:

```
packages:

  "$(match_package)"

     package_policy => "add",
     package_method => xyz;
```

Notes:

This decides what fate is intended for the named package.

- add — Ensure that a package is present (this is the default setting from 3.3.0).
- delete — Ensure that a package is not present.
- reinstall — Delete then add package (warning, non-convergent).
- update — Update the package if an update is available (manager dependent).
- addupdate — Equivalent to 'add' if the package is not installed, and 'update' if it is installed.
- patch — Install one or more patches if available (manager dependent).
- verify — Verify the correctness of the package (manager dependent). The promise is kept if the package is installed correctly, not kept otherwise. Requires setting `package_verify_command`.

7.22.4 package_select

Type: (menu option)

Allowed input range:

```
               >
               <
               ==
               !=
               >=
               <=
```

Synopsis: A criterion for first acceptable match relative to "package_version"

Example:

```
packages:

  "$(exact_package)"

     package_policy => "add",
     package_method => xyz,
     package_select => ">=",
     package_architectures => { "x86_64" },
     package_version => "1.2.3-456";
```

Notes:

This selects the operator that compares the promiser to the state of the system packages currently installed. If the criterion matches, the policy action is scheduled for promise-keeping.

7.22.5 package_version

Type: string
Allowed input range: (arbitrary string)
Synopsis: Version reference point for determining promised version
Example:

```
packages:

  "mypackage"

     package_policy => "add",
     package_method => rpm,
     package_select => "==",
     package_version => "1.2.3";
```

Notes:

Used for specifying the targeted package version when the version is written separately from the name of the command.

7.23 processes promises in 'agent'

Process promises refer to items in the system process table. Note that this is not the same as commands (which are instructions that CFEngine will execute). A process is a command in some state of execution (with a Process Control Block). Promiser objects here are patterns that match line fragments in the system process table (that is, the patterns are unanchored, see Section 2.11.4 [Anchored

vs. unanchored regular expressions], page 39). *Take care to note that process table formats differ between operating systems, and the use of simple patterns such as program-names is recommended. For more sophisticated matches, users should use the* `process_select` *feature.* For example, on many systems, the process pattern `"^cp"` may not match any processes, even though `"cp"` is running! This is because the process table entry may list `"/bin/cp"`. However, the process pattern `"cp"` will also match a process containing `"scp"`, so take care not to oversimplify your patterns (the PCRE pattern anchors `"\b"` and `"\B"` may prove very useful to you).

```
processes:

  "regex contained in process line"

      process_select => process_filter_body,
      restart_class => "activation class for process",
      ..;
```

In CFEngine 2 there was a restart clause for directly executing a command to restart a process. In CFEngine 3 there is instead a class to activate. You must then desribe a `command` in that class to restart the process.

```
commands:

  restart_me::

   "/path/executable" ... ;
```

This rationalizes complex restart-commands and avoids unnecessary overlap between `processes` and `commands`.

The `process_stop` is also arguably a command, but it should be an ephemeral command that does not lead to a persistent process. It is intended only for commands of the form '`/etc/inetd service stop`', not for processes that persist. Processes are restarted at the end of a bundle's execution, but stop commands are executed immediately.

Note: `process_select` was previously called process `filters` in CFEngine 2 and earlier.

```
bundle agent example
{
processes:

 ".*"

    process_count   => anyprocs,
    process_select  => proc_finder;

reports:
```

```
 any_procs::

   "Found processes out of range";
}

########################################################

body process_select proc_finder

{
# Processes started between 5.5 hours and 20 minutes ago
stime_range => irange(ago(0,0,0,5,30,0),ago(0,0,0,0,20,0));
process_result => "stime";
}

########################################################

body process_count anyprocs

{
match_range => "0,0";
out_of_range_define => { "any_procs" };
}
```

In CFEngine 3 we have

```
processes
commands
```

so that there is a clean separation between detection (promises about the process table) and certain repairs (promises to execute commands that start processes). Let's see why.

Executions are about jobs, services, scripts etc. They are properties of an executable file. The referring 'promiser' is a file object. On the other hand a process is a property of a "process identifier" which is a kernel instantiation, a quite different object altogether. So it makes sense to say that

- A "PID" (which is not an executable) promises to be reminded of a signal, e.g.

  ```
  kill signal pid
  ```

- An "command" promises to start or stop itself with a parameterized specification.

  ```
  exec command argument1 argument2 ...
  ```

Neither the file nor the pid necessarily promise to respond to these activations, but they are nonetheless physically meaningful phenomena or attributes associated with these objects.

- Executable files do not listen for signals as they have no active state.
- PIDs do not run themselves or stop themselves with new arguments, but they can use signals as they are running.

Executions lead to processes for the duration of their lifetime, so these two issues are related, although the promises themselves are not.

Services verus processes:

A service is an abstraction that requires processes to run and files to be configured. It makes a lot of sense to wrap services in modular bundles. Starting and stopping a service can be handled in at least two ways. Take the web service as an example.

We can start the service by promising an execution of a daemon (e.g. `httpd`). Normally this execution does not terminate without intervention. We can terminate it in one of two ways:

- Using a process signal, by promising a signal to processes matching a certain pid search
- Using an execution of a termination command, e.g. '`/etc/init.d/apache stop`'.

The first case makes sense if we need to qualify the termination by searching for the processes. The processes section of a CFEngine 3 policy includes a control promise to search for matching processes. If matches are found, signals can be sent to precisely each specific process.

Classes can also be defined, in principle triggering an execution of the stop script, but then the class refers only to the presence of matching pids, not to the individual pids concerned. So it becomes the responsibility of the execution to locate and interact with the pids necessary.

Want it running?:

How do we say simply that we want a service running? In the agent control promises, we could check each service individually.

```
bundlesequence => { Update, Service("apache"), Service("nfsd") };
```

or

```
bundlesequence => { Update, @(globals.all_services)  };
```

The bundle for this can look like this:

```
bundle agent Service(service")
{
processes:

  "$(service)"

      process_count => up("$(service)");

commands:

   "$daemons[$(service)]"

      ifvarclass => "$(service)_up",
      args       => "$args[$(service)]";

}
```

An alternative would be self-contained:

```
bundle agent Service
{
vars:
```

```
  "service" slist => { "apache", "nfsd", "bind" };

processes:

  "$(service)"

      process_count => up("$(service)");

commands:

   "$daemons[$(service)]"

      ifvarclass => "$(service)_up",
      args       => "$args[$(service)]";

}

######################
# Parameterized body
######################

body process_count up("$(s)")

{
match_range => "[0,10]";
out_of_range_define => "$(s)_up";
}
```

7.23.1 process_count (body template)

Type: (ext body)

'in_range_define'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of classes to define if the matches are in range

Example:

```
body process_count example
{
in_range_define => { "class1", "class2" };
}
```

Notes:

Classes are defined if the processes that are found in the process table satisfy the promised process count, i.e. if the promise about the number of processes matching the other criteria is kept.

'match_range'

Type: irange [int,int]

Allowed input range: 0,99999999999

Synopsis: Integer range for acceptable number of matches for this process

Example:

```
body process_count example
{
match_range => irange("10","50");
}
```

Notes:

This is a numerical range for the number of occurrences of the process in the process table. As long as it falls within the specified limits, the promise is considered kept.

'out_of_range_define'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of classes to define if the matches are out of range

Example:

```
body process_count example(s)
{
out_of_range_define => { "process_anomaly", "anomaly_$(s)"};
}
```

Notes:

Classes to activate remedial promises conditional on this promise failure to be kept.

7.23.2 process_select (body template)

Type: (ext body)

'command' **Type**: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression matching the command/cmd field of a process

Example:

```
body process_select example

{
command => "cf-.*";

process_result => "command";
}
```

Notes:

This expression should match the entire COMMAND field of the process table (not just a fragment). This field is usually the last field on the line and thus starts with the first non-space character and ends with the end of line.

'pid' **Type**: irange [int,int]

Allowed input range: 0,99999999999

Synopsis: Range of integers matching the process id of a process

Example:

```
body process_select example
{
pid => irange("1","10");
process_result => "pid";
}
```

Notes:

'pgid' **Type**: irange [int,int]

Allowed input range: 0,99999999999

Synopsis: Range of integers matching the parent group id of a process

Example:

```
body process_select example
{
pgid => irange("1","10");
process_result => "pgid";
}
```

Notes:

'ppid' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the parent process id of a process

Example:

```
body process_select example
{
ppid => irange("407","511");
process_result => "ppid";
}
```

Notes:

'priority' **Type**: irange [int,int]

Allowed input range: `-20,+20`

Synopsis: Range of integers matching the priority field (PRI/NI) of a process

Example:

```
body process_select example
{
priority => irange("-5","0");
}
```

Notes:

'`process_owner`'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of regexes matching the user of a process

Example:

```
body process_select example
{
process_owner => { "wwwrun", "nobody" };
}
```

Notes:

Regular expression should match a legal user name on the system. The regex must match the entire name (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'`process_result`'

Type: string

Allowed input range: `[(process_owner|pid|ppid||pgid|rsize|vsize|status|command|ttime|stime|t`

Synopsis: Boolean class expression returning the logical combination of classes set by a process selection test

Example:

```
body process_select proc_finder(p)

{
process_owner  => { "avahi", "bin" };
command        => "$(p)";
pid            => irange("100","199");
vsize          => irange("0","1000");
process_result => "command.(process_owner|vsize).!pid";
}
```

Notes:

A logical combination of the process selection classifiers. The syntax is the same as that for class expressions. There should be no spaces in the expressions.

'rsize' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the resident memory size of a process, in kilobytes

Example:

```
body process_select
{
rsize => irange("4000","8000");
}
```

Notes:

'status' **Type**: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression matching the status field of a process

Example:

```
body process_select example
{
status => "Z";
}
```

Notes:

For instance, characters in the set '`NRS<sl+..`'. Windows processes do not have status fields.

'stime_range'

Type: irange [int,int]

Allowed input range: `0,2147483647`

Synopsis: Range of integers matching the start time of a process

Example:

```
body process_select example
{
```

```
stime_range => irange(ago(0,0,0,1,0,0),now);
}
```

Notes:

The calculation of time from process table entries is sensitive to Daylight Savings Time (Summer/Winter Time) so calculations could be a hour off. This is for now a bug to be fixed.

'ttime_range'

Type: irange [int,int]

Allowed input range: 0,2147483647

Synopsis: Range of integers matching the total elapsed time of a process

Example:

```
body process_select example
{
ttime_range => irange(0,accumulated(0,1,0,0,0,0));
}
```

Notes:

This is total accumulated time for a process.

'tty'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression matching the tty field of a process

Example:

```
body process_select example
{
tty => "pts/[0-9]+";
}
```

Notes:

Windows processes are not regarded as attached to any terminal, so they all have tty '?'.

'threads' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the threads (NLWP) field of a process

Example:

```
body process_select example
{
threads => irange(1,5);
}
```

Notes:

'vsize' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the virtual memory size of a process, in kilobytes

Example:

```
body process_select example
{
vsize => irange("4000","9000");
}
```

Notes:

On Windows, the virtual memory size is the amount of memory that cannot be shared with other processes. In Task Manager, this is called Commit Size (Windows 2008), or VM Size (Windows XP).

7.23.3 process_stop

Type: string

Allowed input range: `"?(/.*)`

Synopsis: A command used to stop a running process

Example:

```
processes:
```

```
 "snmpd"

        process_stop => "/etc/init.d/snmp stop";
```

Notes:

As an alternative to sending a termination or kill signal to a process, one may call a 'stop script' to perform a graceful shutdown.

7.23.4 restart_class

Type: string
Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`
Synopsis: A class to be defined globally if the process is not running, so that a command: rule can be referred to restart the process
Example:

```
processes:

   "cf-serverd"

     restart_class => "start_cfserverd";

commands:

  start_cfserverd::

    "/var/cfengine/bin/cf-serverd";
```

Notes:

This is a signal to restart a process that should be running, if it is not running. Processes are signalled first and then restarted later, at the end of bundle execution, after all possible corrective actions have been made that could influence their execution.

Windows does not support that processes start themselves in the background, like Unix daemons usually do (i.e. fork off a child process). Therefore, it may be useful to specify an action bodypart that sets background to true in a commands promise that is invoked by the class set by restart_class. See the commands promise type for more information.

7.23.5 signals

Type: (option list)

Allowed input range:

```
hup
int
trap
kill
pipe
cont
abrt
stop
quit
term
child
usr1
usr2
bus
segv
```

Synopsis: A list of menu options representing signals to be sent to a process

Example:

```
processes:

 cfservd_out_of_control::

   "cfservd"

        signals         => { "stop" , "term" },
        restart_class   => "start_cfserv";

 any::

   "snmpd"

        signals         => { "term" , "kill" };
```

Notes:

Signals are presented as an ordered list to the process. On windows, only the kill signal is supported, which terminates the process.

7.24 services promises in 'agent'

A service is a set of zero or more processes. It can be zero if the service is not currently running. Services run in the background, and do not require user intervention.

Service promises may be viewed as an abstraction of process and commands promises. An important distinguiser is however that a single service may consist of multiple processes. Additionally, services are registered in the operating system in some way, and gets a unique name. Unlike processes and commands promises, this makes it possible to use the same name both when it is running and not.

Some operating systems are bundled with a lot of unused services that are running as default. At the same time, faulty or inherently insecure services are often the cause of security issues. With CFEngine, one can create promises stating services that should be stopped and disabled.

The operating system may start a service at boot time, or it can be started by CFEngine. Either way, CFEngine will ensure that the service maintains the correct state (started, stopped, or disabled). On some operating systems, CFEngine also allows services to be started on demand, i.e. when they are needed. This is implemented though the `inetd` or `xinetd` daemon on Unix. Windows does not support this.

CFEngine also allows for the concept of dependencies between services, and can automatically start or stop these, if desired. Parameters can be passed to services that are started by CFEngine.

```
bundle agent example
{
services:

  "Dhcp"
    service_policy => "start",
    service_dependencies => { "Alerter", "W32Time" },
    service_method => winmethod;
}

########################################################

body service_method winmethod
{
  service_type => "windows";
  service_args => "--netmask=255.255.0.0";
  service_autostart_policy => "none";
  service_dependence_chain => "start_parent_services";
}
```

Services promises for Windows are only available in CFEngine Nova and above. Windows Vista/Server 2008 and later introduced new complications to the service security policy. Therefore,

when testing `services` promises from the command line, CFEngine may not be given proper access rights, which gives errors like "Access is denied". However, when running through the CFEngine Nova Executor service, typical for on production machines, CFEngine has sufficent rights.

Services of type '`generic`' promises are implemented for all operating systems and are merely as a convenient front-end to `processes` and `commands`. If nothing else is specified, CFEngine looks for an special reserved agent bundle called

```
bundle agent standard_services(service,state)
{
...
}
```

This bundle is called with two parameters: the name if the service and a start/stop state variable. The CFEngine standard library defines many common services for standard operating systems for convenience. If no `service_bundle` is defined in a `service_method` body, then CFEngine assumes the '`standard_services`' bundle to be the default source of action for the services. This is executed just like a `methods` promise on the service bundle, so this is merely a front-end.

The standard bundle can be replaced with another, as follows, e.g. for testing purposes:

```
body common control
{
bundlesequence => { "test" };
}

#

bundle agent test
{
vars:

 "mail" slist => { "spamassassin", "postfix" };


services:

  "www" service_policy => "start",
        service_method => service_test;


  "$(mail)" service_policy => "stop",
        service_method => service_test;
}

#

body service_method service_test
{
service_bundle => non_standard_services("$(this.promiser)","$(this.service_policy)");
}
```

```
#

bundle agent non_standard_services(service,state)
{
reports:

  !done::

    "Test service promise for \"$(service)\" -> $(state)";
}
```

Note that the special variables `$(this.promiser)` and `$(this.service_policy)` may be used to fill in the service and state parameters from the promise definition. The `$(this.service_policy)` variable is only defined for services promises.

7.24.1 service_policy

Type: (menu option)

Allowed input range:

```
               start
               stop
               disable
               restart
               reload
```

Synopsis: Policy for cfengine service status

Example:

```
services:

  "Telnet"
     service_policy => "disable";
```

Notes:

If set to `start`, CFEngine Nova will keep the service in a running state, while `stop` means that the service is kept in a stopped state. `disable` implies `stop`, and ensures that the service can not be started directly, but needs to be enabled somehow first (e.g. by changing file permissions).

7.24.2 service_dependencies

Type: slist

Allowed input range: `[a-zA-Z0-9_$(){}\[\].:]+`

Synopsis: A list of services on which the named service abstraction depends

Example:

```
services:

  "ftp"
    service_policy => "start",
    service_dependencies => { "network", "logging" };
```

Notes:

A list of services that must be running before the service can be started. These dependencies can be started automatically by CFEngine Nova if they are not running — see `service_dependence_chain`. However, the dependencies will never be implicitly stopped by CFEngine Nova. Specifying dependencies is optional.

Note that the operating system may keep an additional list of dependencies for a given service, defined during installation of the service. CFEngine Nova requires these dependencies to be running as well before starting the service. The complete list of dependencies is thus the union of `service_dependencies` and the internal operating system list.

7.24.3 `service_method` (body template)

Type: (ext body)

'`service_args`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Parameters for starting the service as command

Example:

```
body service_method example
{
  service_args => "-f filename.conf --some-argument";
}
```

Notes:

These arguments will only be passed if CFEngine Nova starts the service. Thus, set `service_autostart_policy` to `none` to ensure that the arguments are always passed.

Escaped quoutes can be used to pass an argument contianing spaces as a single argument, e.g. "-f \"file name.conf\"". Passing arguments is optional.

'service_autostart_policy'

Type: (menu option)

Allowed input range:

```
none
boot_time
on_demand
```

Synopsis: Should the service be started automatically by the OS

Example:

```
body service_method example
{
  service_autostart_policy => "boot_time";
}
```

Notes:

Defaults to `none`, which means that the service is not registered for automatic startup by the operating system in any way. It must be `none` if `service_policy` is not `start`. `boot_time` means the service is started at boot time, while `on_demand` means that the service is dispatched once it is being used.

`on_demand` is not supported by Windows, and is implemented through inetd or xinetd on Unix.

'service_bundle'

Type: (ext bundle) (Separate Bundle)

'service_dependence_chain'

Type: (menu option)

Allowed input range:

```
ignore
start_parent_services
stop_child_services
all_related
```

Synopsis: How to handle dependencies and dependent services

Example:

```
body service_method example
{
```

```
  service_dependence_chain => "start_parent_services";
}
```

Notes:

The service dependencies include both the dependencies defined by the operating system and in `service_dependencies`, as described there.

Defaults to `ignore`, which means that CFEngine Nova will never start or stop dependencies or dependent services, but fail if dependencies are not satisfied. `start_parent_services` means that all dependencies of the service will be started if they are not already running. When stopping a service, `stop_child_services` means that other services that depend on this service will be stopped also. `all_related` means both `start_parent_services` and `stop_child_services`.

Note that this setting also affects dependencies of dependencies and so on.

For example, consider the case where service A depends on B, which depends on C. If we want to start B, we must first make sure A is running. If `start_parent_services` or `all_related` is set, CFEngine Nova will start A, if it is not running. On the other hand, if we want to stop B, C needs to be stopped first. `stop_child_services` or `all_related` means that CFEngine Nova will stop C, if it is running.

'`service_type`'

Type: (menu option)

Allowed input range:

```
windows
generic
```

Synopsis: Service abstraction type

Example:

```
body service_method example
{
  type => "windows";
}
```

Notes:

On Windows this defaults to, and must be `windows`. Unix systems can however have multiple means of registering services, but the choice must be available on the given system.

7.25 storage promises in 'agent'

Storage promises refer to disks and filesystem properties.

```
storage:

   "/disk volume or mountpoint"

     volume => volume_body,
     ...;
```

In CFEngine 2, storage promises were divided into `disks` or `required`, and `misc_mounts` types. The old mount-models for binary and home servers has been deprecated and removed from CFEngine 3. Users who use these models can reconstruct them from the low-level tools.

```
bundle agent storage

{
storage:

  "/usr" volume  => mycheck("10%");
  "/mnt" mount   => nfs("nfsserv.example.org","/home");

}

#######################################################

body volume mycheck(free)   # reusable template

{
check_foreign  => "false";
freespace      => "$(free)";
sensible_size  => "10000";
sensible_count => "2";
}

body mount nfs(server,source)

{
mount_type => "nfs";
mount_source => "$(source)";
mount_server => "$(server)";
edit_fstab => "true";
```

```
}
```

7.25.1 mount (body template)

Type: (ext body)

'edit_fstab'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false add or remove entries to the file system table ("fstab")

Default value: false

Example:

```
body mount example
{
edit_fstab => "true";
}
```

Notes:

The default behaviour is to not place edits in the file system table.

'mount_type'

Type: (menu option)

Allowed input range:

```
nfs
nfs2
nfs3
nfs4
```

Synopsis: Protocol type of remote file system

Example:

```
body mount example
{
mount_type => "nfs3";
}
```

Notes:

This field is mainly for future extensions.

'mount_source'

Type: string

Allowed input range: "?(/.*)

Synopsis: Path of remote file system to mount

Example:

```
body mount example
{
mount_source "/location/disk/directory";
}
```

Notes:

This is the location on the remote device, server, SAN etc.

'mount_server'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Hostname or IP or remote file system server

Example:

```
body mount example
{
mount_server => "nfs_host.example.org";
}
```

Notes:

Hostname or IP address, this could be on a SAN.

'mount_options'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of option strings to add to the file system table ("fstab")

Example:

```
body mount example
{
mount_options => { "rw", "acls" };
}
```

Notes:

This list is concatenated in a form appropriate for the filesystem. The options must be legal options for the system mount commands.

'unmount' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false unmount a previously mounted filesystem

Default value: false

Example:

```
body mount example
{
unmount => "true";
}
```

Notes:

7.25.2 volume (body template)

Type: (ext body)

'check_foreign'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false verify storage that is mounted from a foreign system on this host

Default value: false

Example:

```
body volume example

{
#..
check_foreign  => "false";
}
```

Notes:

CFEngine will not normally perform sanity checks on filesystems which are not local to the host. If true it will ignore a partition's network location and ask the current host to verify storage located physically on other systems.

'freespace'

Type: string

Allowed input range: [0-9]+[MBkKgGmb%]

Synopsis: Absolute or percentage minimum disk space that should be available before warning

Example:

```
body volume example1
{
freespace => "10%";
```

```
}

body volume example2
{
freespace => "50M";
}
```

Notes:

The amount of freespace that is promised on a storage device. Once this promise is found not to be kept (that is, if the free space falls below the promised value), warnings are generated. You may also want to use the results of this promise to control other promises, See Section 6.6.2 [classes in *], page 179.

'sensible_size'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Minimum size in bytes that should be used on a sensible-looking storage device

Example:

```
body volume example
{
sensible_size => "20K";
}
```

Notes:

```
body volume control
{
sensible_size => "20K";
}
```

'sensible_count'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Minimum number of files that should be defined on a sensible-looking storage device

Example:

```
body volume example
{
sensible_count => "20";
}
```

Notes:

Files must be readable by the agent, i.e. it is assumed that the agent has privileges on volumes being checked.

'scan_arrivals'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false generate pseudo-periodic disk change arrival distribution

Default value: false

Example:

```
body volume example
{
scan_arrivals => "true";
}
```

Notes:

This operation should not be left 'on' for more than a single run (maximum once per week). It causes CFEngine to perform an extensive disk scan noting the schedule of changes between files. This can be used for a number of analyses including optimum backup schedule computation.

8 Bundles of server

```
bundle server access_rules()

{
access:

  "/home/mark/PrivateFiles"

    admit   => { ".*\.example\.org" };

  "/home/mark/\.cfagent/bin/cf-agent"

    admit   => { ".*\.example\.org" };

roles:

  ".*"  authorize => { "mark" };
}
```

Bundles in the server describe access promises on specific file and class objects supplied by the server to clients.

8.1 access promises in 'server'

Access promises are conditional promises made by the server about file objects. The promise has two consequences. For file copy requests, the file becomes transferrable to the remote client according to the conditions specified in the server promse (i.e. if the connection encryption requirements are met, and if the client has been granted appropriate privileges with maproot (like its NFS counterpart) to be able to see file objects not owned by the server process owner).

The promise has two mutally exclusive attributes 'admit' and 'deny'. Use of 'admit' is preferred as mistakes and omissions can easily be made when excluding from a group.

When access is granted to a directory, the promise is automatically given about all of its contents and sub-directories. The access promise allows overlapping promises to be made, and these are kept in a first-come-first-served fashion. Thus file objects (promisers) should be listed in order of most-specific file first. In this way, specific promises will override less specific ones.

```
access:

   "/path/file_object"

     admit   => { "hostname", "ipv4_address", "ipv6_address"  };
```

Example:

```
#########################################################
# Server config
#########################################################

body server control

{
allowconnects         => { "127.0.0.1" , "::1" };
allowallconnects      => { "127.0.0.1" , "::1" };
trustkeysfrom         => { "127.0.0.1" , "::1" };
}

#########################################################

bundle server access_rules()

{
access:

  "/source/directory"
          comment => "Access to file transfer",
          admit   => { "127.0.0.1" };

  # Grant orchestration communication

  "did.*"
          comment => "Access to class context (enterprise)",
    resource_type => "context",
            admit => { "127.0.0.1" };


  "value of my test_scalar, can expand variables here - $(sys.host)"
          comment => "Grant access to the string in quotes, by name test_scalar",
           handle => "test_scalar",
    resource_type => "literal",
            admit => { "127.0.0.1" };
```

```
  "XYZ"
          comment => "Grant access to contents of persistent scalar variable XYZ",
    resource_type => "variable",
            admit => { "127.0.0.1" };

  # Client grants access to CFEngine hub access

  "delta"
    comment => "Grant access to cfengine hub to collect report deltas",
    resource_type => "query",
          admit   => { "127.0.0.1"  };
  "full"
          comment => "Grant access to cfengine hub to collect full report dump",
    resource_type => "query",
          admit   => { "127.0.0.1"  };

  policy_hub::

  "collect call"
          comment => "Grant access to cfengine client to request the collection of its reports",
    resource_type => "query",
          admit   => { "10.1.2.*" };


}
```

Entries may be literal addresses of IPv4 or IPv6, or any name registered in the POSIX `gethostbyname` service.

8.1.1 admit

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host names or IP addresses to grant access to file objects

Example:

```
access:

  "/home/mark/LapTop"

    admit   => { "127.0.0.1", "192.168.0.1/24", ".*\.domain\.tld"  };
```

Notes:

Admit promises grant access to file objects on the server. Arguments may be IP addresses or hostnames, provided DNS name resolution is active. In order to reach this stage, a client must first have passed all of the standard connection tests in the control body.

The lists may contain network addresses in CIDR notation or regular expressions to match the IP address or name of the connecting host.

8.1.2 deny

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host names or IP addresses to deny access to file objects

Example:

```
bundle server access_rules()

{
access:

  "/path"

    admit   => { ".*\.example\.org" },
    deny    => { "badhost_1\.example\.org", "badhost_1\.example\.org" };
}
```

Notes:

Denial is for special exceptions. A better strategy is always to grant on a need to know basis. A security policy based on exceptions is a weak one.

Note that only regular expressions or exact matches are allowed in this list, as non-specific matches are too greedy for denial.

8.1.3 maproot

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host names or IP addresses to grant full read-privilege on the server

Example:

```
access:
```

```
"/home"

      admit => { "backup_host.example.org" },
ifencrypted => "true",

    # Backup needs to have access to all users

    maproot => { "backup_host.example.org" };
```

Notes:

Normally users authenticated by the server are granted access only to files owned by them and no-one else. Even if the `cf-serverd` process runs with root privileges on the server side of a client-server connection, the client is not automatically granted access to download files owned by non-privileged users. If `maproot` is true then remote `root` users are granted access to all files.

A typical case where mapping is important is in making backups of many user files. On the Windows `cf-serverd`, `maproot` is required to read files if the connecting user does not own the file on the server.

8.1.4 ifencrypted

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Default value: false

Synopsis: true/false whether the current file access promise is conditional on the connection from the client being encrypted

Example:

```
access:

   "/path/file"

    admit     => { ".*\.example\.org" },
    ifencrypted => "true";
```

Notes:

If this flag is true a client cannot access the file object unless its connection is encrypted.

8.1.5 resource_type

Type: (menu option)
Allowed input range:

```
path
literal
context
query
variable
```

Synopsis: The type of object being granted access (the default grants access to files)
Example:

```
bundle server access_rules()

{
access:

  "value of my test_scalar, can expand variables here - $(sys.host)"
    handle => "test_scalar",
    comment => "Grant access to contents of test_scalar VAR",
    resource_type => "literal",
    admit => { "127.0.0.1" };

  "XYZ"
    resource_type => "variable",
    handle => "XYZ",
    admit => { "127.0.0.1" };


  # On the policy hub

  "collect_calls"
     resource_type => "query",
           admit   => { "127.0.0.1" };

  # On the isolated client in the field
```

```
 "delta"
    comment => "Grant access to cfengine hub to collect report deltas",
    resource_type => "query",
          admit   => { "127.0.0.1"  };
  "full"
          comment => "Grant access to cfengine hub to collect full report dump",
    resource_type => "query",
          admit   => { "127.0.0.1"  };


}
```

Notes:

By default, access to resources granted by the server are files. However, sometimes it is useful to cache `literal` strings, hints and data in the server, e.g. the contents of variables, hashed passwords etc for easy access. In the case of literal data, the promise handle serves as the reference identifier for queries. Queries are instigated by function calls by any agent.

If the resource type is `literal`, CFEngine will grant access to a literal datastring. This string is defined either by the promiser itself, but the name of the variable is the identifier given by the promise handle of the access promise, since the promiser string might be complex.

If the resource type is `variable` then the promiser is the name of a persistent scalar variable defined on the server-host. Currently persistent scalars are only used internally by Enterprise CFEngine to hold enumerated classes for orchestration purposes.

If you want to send the value of a policy defined variable in the server host (which for some reason is not available directly through policy on the client, e.g. because they have differnet policies), then you could use the construction

```
access:

  "$(variable_name)"

         handle => "variable_name",
  resource_type => "literal";
```

If the resource type is `context`, the promiser is treated as a regular expression to match persistent classes defined on the server host. If these are matched by the request from the client, they will be transmitted, See Section 11.86 [Function remoteclassesmatching], page 493.

The term `query` may also be used in commercial versions of CFEngine to query the server for data from embedded databases. This is currently for internal use only, and is used to grant access to report 'menus'. If the promiser of a query request is called '`collect_calls`', this grants access to server peering collect-call tunneling, See Section 5.3.7 [call_collect_interval in server], page 98.

8.2 `roles` promises in '`server`'

Roles promises are server-side decisions about which users are allowed to define soft-classes on the server's system during remote invocation of `cf-agent`. This implements a form of Role Based Access Control (RBAC) for pre-assigned class-promise bindings. The user names cited must be attached to trusted public keys in order to be accepted. The regular expression must match the entire name (that is, they are anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

```
roles:

  "regex"

     authorize => { "usernames", ... };
```

It is worth re-iterating here that it is not possible to send commands or modify promise definitions by remote access. At best users may try to send classes when using `cf-runagent` in order to activate sleeping promises. This mechanism limits their ability to do this.

```
bundle server access_rules()

{
roles:

  # Allow mark

  "Myclass_.*"  authorize => { "mark" };
}
```

In this example user '`mark`' is granted permission to remotely activate classes matching the regular expression when '`Mark_.*`' using the `cf-runagent` to activate CFEngine. In this way one can implement a form of Role Based Access Control (RBAC), provided users do not have privileged access on the host directly.

8.2.1 `authorize`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of public-key user names that are allowed to activate the promised class during remote agent activation

Example:

```
roles:
```

```
".*"  authorize => { "mark", "marks_friend" };
```

Notes:

Part of Role Based Access Control (RBAC) in CFEngine. The users listed in this section are granted access to set certain classes by using the remote `cf-runagent`. The user-names will refer to public key identities already trusted on the system.

9 Bundles of knowledge

```
bundle knowledge system

{
topics:

 Troubleshooting::

  "Segmentation fault"
       association => a("is caused by","Bad memory reference","can cause");

  "Remote connection problem";
  "Web server not running";
  "Print server not running";
  "Bad memory reference";
}
```

Knowledge bundles describe topic maps, i.e. Topics, Associations and Occurrences (of topics in documents). This is for knowledge modelling and has no functional effect on a system.

9.1 inferences promises in 'knowledge'

```
inferences:

  "is close to"
          comment => "Cluster property",
        precedent => { "is close to" },
        qualifier => { "is close to" };

  "is far from"
          comment => "Remote cluster property",
        precedent => { "is far from" },
        qualifier => { "is close to" };
```

History: Was introduced in version 3.1.0

Inference promises are used to perform simple contextual reasoning in the knowledge map. This feature is currently only supported in commercial versions of CFEngine as it is developed.

The promiser of an inference promise is the *result* of the inference, i.e. the conclusion to be drawn from combining two knowledge assertions. The body specifies what existing associations must be in place between topics in order to draw the conclusion between the start and the end.

```
          precedent                 qualifier
TOPIC 1 --------------> TOPIC 2 ---------------> TOPIC 3

                  promised inference
TOPIC 1 ---------------------------------------> TOPIC 3
```

For example,

```
       is mother to          is married to
ALICE --------------> BOB -----------------> CAROL

              is mother in law to
ALICE ---------------------------------------> CAROL
```

Note that, like all promises, they are expected to be unique. Multiple promisers promising different bodies is potentially inconsistent. However, inference is inherently ambiguous and we need to accomodate multiple patterns. To this end, lists of regular expressions may be used to match multiple instances.

9.1.1 precedents

Type: slist

Allowed input range: (arbitrary string)

Synopsis: The foundational vector for a trinary inference

Example:

```
inferences:

  "is far from"
           comment => "Remote cluster property",
         precedent => { "is far from"},
         qualifier => { "is close to", "is far from" };
```

Notes:

History: Was introduced in version 3.1.0b3,Nova 2.0.0b1 (2010)

A general regular expression may be used to match suitable alternatives, so as to make the promise unique.

9.1.2 qualifiers

Type: slist

Allowed input range: (arbitrary string)

Synopsis: The second vector in a trinary inference

Example:

```
inferences:

  "is far from"
           comment => "Remote cluster property",
         precedent => { "is far from" },
         qualifier => { "is close to|is far from" };
```

Notes:

History: Was introduced in version 3.1.0 (2010)

A general regular expression may be used to match suitable alternatives, so as to make the promise unique.

9.2 things promises in 'knowledge'

History: Was introduced in version 3.2, Nova 2.1 (2011)

Things are a special subset of topics that behave like objects in the world. We have separated out things from more abstract topics to make it easier to talk about the. Things are typically objects we make inventories of, that influence one another and might be connected. We interact with things in IT management much more concretely than we do with abstract topics.

To make it simpler to talk about things, we there introduce things promises.

```
body knowledge TheRealWorld
{
things:

  networks::

     "10.20.30.40"  is_connected_to => { "router 46", "computers::computer 23" };

  computers::

     "computer 23" belongs_to => { "Phileas Phogg", "ACME punchcard agency" };

}
```

Things promises are in every way equivalent to the more general topics promises. Things can be extended as topics. The contexts are interchangeable between things and topics. The only purpose of things is to make plainer a description of the 'physical' configurations of regular worldly things.

9.2.1 synonyms

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of words to be treated as equivalents in the defined context

Example:

```
 mathematics::

   "tree" synonyms => { "DAG", "directed acyclic graph" };
```

Notes:

History: Was introduced in version 3.1.3a1,Nova 2.0.2a1 (2010)

This may be used to simplify the identification of synonyms during topic searches.

9.2.2 affects

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

   "The Moon"   affects => {  "surf", "tides" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

9.2.3 belongs_to

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

  "router-123"

           comment => "Located at 23 Marlborough Street",
        belongs_to => { "company::cfengine" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

9.2.4 causes

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
bundle knowledge test
{
things:

  "program crash" causes => { "rootprocs_low_anomaly" },
                    certainty => "possible";

}
```

Notes:

History: Was introduced in version 3.2.0, Nova 2.1.0 (2011)
The complement of 'is_caused_by' for convenience.

9.2.5 certainty

Type: (menu option)
Allowed input range:

```
certain
uncertain
possible
```

Synopsis: Selects the level of certainty for the proposed knowledge, for use in inferential reasoning
Example:

```
bundle knowledge test
{
things:

  "router one" is_connected_to => { "computer one" },
                     certainty => "uncertain";

  "router"             affects => { "network services" },
                     certainty => "possible";
}
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1 (2011)

Certainty is used in automated reasoning about knowledge. It modifies the relationships between things. For example, the certain relationship 'is affected by' would become 'can be affected by' (possible) or 'might be affected by' (uncertain).

9.2.6 `determines`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

  "router one" determines => { "network connectivity" },
                certainty => "uncertain";
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1 (2011)

9.2.7 `generalizations`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of words to be treated as super-sets for the current topic, used when reasoning

Example:

```
 topics:
```

```
  persons::

     "mark"  generalizations => { "person", "staff", "human being" };

  any::

     "10.10.10.10/24" generalizations => { "network", "CIDR format" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

Generalizations are ways of thinking about topics in more general terms. They are somewhat like container 'types' or 'classes' in hierarchical modelling, but they need not be mutually exclusive categories.

Generalizations may be used in topic-lifting, a kind of brain-storming about issues, when searching for diagnostic explanations.

9.2.8 implements

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

   "my promise"
              implements => { "my goal" };
```

Notes:

History: Was introduced in 3.4.0

9.2.9 involves

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

   "desired state"
```

```
      involves => {
                   "business goals",
                   "knowing what state to configure",
                   "knowing what objects to maintain"
                   };
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

9.2.10 `is_caused_by`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
bundle knowledge test
{
things:

  "core dump" is_caused_by => { "memory fault" },
                     certainty => "certain";

}
```

Notes:

History: Was introduced in version 3.2.0, Nova 2.1.0 (2011)
The complement of ‘causes’ for convenience of expression.

9.2.11 `is_connected_to`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

 networks::
```

```
  "192.23.45.0/24"         comment => "Secure network, zone 0. Single octet for corporate offices",
                   is_connected_to => { "oslo-hub-123" };

  "192.12.74.0/23"         comment => "Zone 1, double octet for the London office developer network"
                   is_connected_to => { "oslo-hub-123" };

  "192.12.74.0/23"         comment => "Secure, single octet for the NYC office",
                   is_connected_to => { "nyc-hub-456" };
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1.0 (2011)

9.2.12 `is_determined_by`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

  "why" is_determined_by => { "bodyparts::comment", "Semantic commentary" };
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)
This is the inverse of `determines`.

9.2.13 `is_followed_by`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

  "Installing CFEngine"

    is_followed_by => { "bootstrapping", "policy editing" };
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

9.2.14 `is_implemented_by`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

   "my goal"
             is_implemented_by => { "my promise" };
```

Notes:

History: Was introduced in 3.4.0

9.2.15 `is_located_in`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

 countries::

   "UK"                 synonyms => { "Great Britain" },
                   is_located_in => { "EMEA", "Europe" };

   "Singapore"     is_located_in => { "APAC", "Asia" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

9.2.16 is_measured_by

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

 service_measurements::

  "login services" is_measured_by => { "ssh_in" };
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0

9.2.17 is_part_of

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

  "host 1" is_part_of => { "123.456.789.0/24" };
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1 (2011)

9.2.18 is_preceded_by

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

  "disk failure"
```

```
    is_preceded_by => { "write errors", "read errors" },
         certainty => "possible";
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

9.2.19 measures

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

 service_measurements::

  "ssh_in"     measures => { "services::login services" },
               measures => { "ssh" };
```

Notes:

History: Was introduced in 3.4.0, Enterprise 3.0

9.2.20 needs

Type: slist
Allowed input range: (arbitrary string)
Synopsis: Special fixed relation for describing topics that are things
Example:

```
things:

  "rack 123"     needs => { "power", "cooling" };

  "computer"     needs => { "cleaning", "monitoring" },
             certainty => "possible";
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1 (2011)

9.2.21 provides

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

   "host 23" provides => { "www", "email" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

9.2.22 uses

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Special fixed relation for describing topics that are things

Example:

```
things:

  "apache 2.3" uses => { "mysql 4.5" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

9.3 topics promises in 'knowledge'

Topic promises are part of the knowledge management engine. A topic is any string that refers to a concept or subject that we wish to include in a knowledge base. If a topic has a very long name, it is best to made the promiser object a short name and use the `comment` field to add the long explanation (e.g. unique acronym and full text).

```
topics:

 "topic string"

  comment => "long name..",
  ...;
```

Topics form associative structures based entirely on an abstract space of natural language. Actually, this is only slightly more abstract than files, processes and commands etc. The main difference in knowledge management is that there are no corrective or maintenance operations associated with knowledge promises.

Class membership in knowledge management is subtly different from other parts of CFEngine. If a topic lies in a certain class context, the topic uses it as a type-label. This is used for disambiguation of subject-area in searches rather than for disambiguation of rules between physical environments.

```
bundle knowledge example
{
topics:

   "Distro"
      comment     => "Distribution of linux",
      association => a("is a packaging of","Linux","is packaged as a");
}
```

Topics are basically identifiers, where the comment field here is a long form of the subject string. Associations form semantic links between topics. Topics can appear multiple times in order to form multiple associations.

9.3.1 `association` (body template)

Type: (ext body)

'`forward_relationship`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of forward association between promiser topic and associates

Example:

```
body association example
{
forward_relation => "is bigger than";
}
```

Notes:

'backward_relationship'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of backward/inverse association from associates to promiser topic

Example:

```
body association example
{
# ..
backward_relationship => "is less than";
}
```

Notes:

Denotes the inverse name which is used to 'moralizing' the association graph.

'associates'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of associated topics by this forward relationship

Example:

```
body association example(literal,scalar,list)

{
#...
associates => { "literal", $(scalar),  @(list)};
}
```

Notes:

An element of an association which is a list of topics to which the current topic is associated.

9.3.2 synonyms

Type: slist
Allowed input range: (arbitrary string)
Synopsis: A list of words to be treated as equivalents in the defined context
Example:

```
 mathematics::

   "tree" synonyms => { "DAG", "directed acyclic graph" };
```

Notes:

History: Was introduced in version 3.1.3a1,Nova 2.0.2a1 (2010)

This may be used to simplify the identification of synonyms during topic searches.

9.3.3 generalizations

Type: slist
Allowed input range: (arbitrary string)
Synopsis: A list of words to be treated as super-sets for the current topic, used when reasoning
Example:

```
 topics:

  persons::

     "mark"  generalizations => { "person", "staff", "human being" };

  any::

     "10.10.10.10/24" generalizations => { "network", "CIDR format" };
```

Notes:

History: Was introduced in version 3.2, Nova 2.1 (2011)

Generalizations are ways of thinking about topics in more general terms. They are somewhat like container 'types' or 'classes' in hierarchical modelling, but they need not be mutually exclusive categories.

Generalizations may be used in topic-lifting, a kind of brain-storming about issues, when searching for diagnostic explanations.

9.4 occurrences promises in 'knowledge'

Occurrences are documents or information resources that discuss topics. An occurrence promise asserts that a particular document of text resource in fact represents information about one or more topics. This is used to construct references to actual information in a topic map.

```
occurrences:

  topic_name::

    "URL reference or literal string"

        represents => { "sub-topic disambiguator", ... },
        representation => "literal or url";
```

```
Mark_Burgess::

    "http://www.iu.hio.no/~mark"
           represents => { "Home Page" };

lvalue::

    "A variable identifier, i.e. the left hand side of an '=' association. The promiser in a variab
           represents => { "Definitions" },
           representation => "literal";

Editing_Files::

"http://www.cfengine.org/confdir/customizepasswd.html"
  represents => { "Setting up users" };
```

Occurrences are pointers to information about topics. This might be a literal text string or a URL reference to an external document.

9.4.1 about_topics

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of topics that the document or resource addresses

Example:

```
 "/docs/SpecialTopic_RBAC.html#tag"

     represents => { "Text section" },
   about_topics => { "defining roles" };
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

As of CFEngine 3.3.0, this represents a list of topics on which the named occurrence provides information.

History: Previously the context class had been used for this purpose, but this led to confusion between usage or context and the subjects covered by a document.

9.4.2 `represents`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of explanations for what relationship this document has to the topics it is about
Example:

```
occurrences:

  Promise_Theory::

    "A theory of autonomous actors that offer certainty through promises"

      represents     => { "Definitions" },
      representation => "literal";
```

Notes:

The sub-topic or occurrence-type represented by the document reference in a knowledge base. This string is intended as an annotation to the reader about the nature of the inforation located in the occurrence document. It should be used 'creatively'.

If the document type is an image and one of the items in this list is a url, beginning with either '/' or '`http`', then `cf-know` treats the reference as a url to be reached when the image is clicked on.

9.4.3 `representation`

Type: (menu option)

Allowed input range:

```
literal
url
db
file
web
image
portal
```

Synopsis: How to interpret the promiser string e.g. actual data or reference to data

Example:

```
occurrences:

  Promise_Theory::

    "A theory of autonomous actors that offer certainty through promises"

      represents     => { "Definitions" },
      representation => "literal";
```

Notes:

The form of knowledge representation in a topic map occurrence reference. If the type `portal` is used it assumes that a new website should open in a new target window.

10 Bundles of monitor

```
bundle monitor example
{
measurements:

     # Discover disk device information

  "/bin/df"

      handle => "free_diskspace_watch",
      stream_type => "pipe",
      data_type => "slist",
      history_type => "static",
      units => "device",
      match_value => file_systems;


}
```

Monitor bundles contain user defined promises for system discovery and monitoring.

10.1 measurements promises in 'monitor'

These features are available only in Enterprise versions of CFEngine.

CFEngine's monitoring component `cf-monitord` records a number of performance data about the system by default. These include process counts, service traffic, load average and cpu utilization and temperature when available.

CFEngine Nova extends this in two ways. First it adds a three year trend summary based any 'shift'-averages. Second, it adds customizable promises to monitor or log very specific user data through a generic interface. The end result is to either generate a periodic time series, like the above mentioned values, or to log the results to custom-defined reports.

CFEngine Nova adds a new promise type in bundles for the monitoring agent. These are written just like all other promises within a bundle destined for the agent concerned (however, you do not need to add them to the `bundlesequence` – they are executed by `cf-monitord` because they are bundles of type `monitor`). In this case:

```
bundle monitor watch

{
measurements:

  # promises ...
```

```
}
```

It is important to specificy a promise `handle` for measurement promises, as the names defined in the handle are used to determine the name of the log file or variable to which data will be reported. Log files are created under '`WORKDIR/state`'. Data that have no history type are stored in a special variable context called '`mon`', analogous to the system variables in '`sys`'. Thus the values may be used in other promises in the form `$(mon.handle)`.

```
  # Follow a special process over time
  # using CFEngine's process cache to avoid resampling

   "/var/cfengine/state/cf_rootprocs"

      handle => "monitor_self_watch",
      stream_type => "file",
      data_type => "int",
      history_type => "weekly",
      units => "kB",
      match_value => proc_value(".*cf-monitord.*",
         "root\s+[0-9.]+\s+[0-9.]+\s+[0-9.]+\s+[0-9.]+\s+([0-9]+).*");


  # Discover disk device information

  "/bin/df"

      handle => "free_diskspace_watch",
      stream_type => "pipe",
      data_type => "slist",
      history_type => "static",
      units => "device",
      match_value => file_systems;
      # Update this as often as possible

}

##########################################################

body match_value proc_value(x,y)
{
select_line_matching => "$(x)";
extraction_regex => "$(y)";
}
```

```
body match_value file_systems
{
select_line_matching => "/.*";
extraction_regex => "(.*)";
}
```

Notes:

The general pattern of these promises is to decide the source of the information either file or pipe, determine the data type (integer, string etc.), specify a pattern to match the result in the file stream and then specify what to do with the result afterwards.

Standard measurements:

The `cf-monitord` service monitors a number of variables as standard on Unix and Windows systems. Windows is fundamentally different from Unix and currently has less support for out-of-the-box probes.

1. users: Users logged in
2. rootprocs: Privileged system processes
3. otherprocs: Non-privileged process
4. diskfree: Free disk on / partition
5. loadavg: % kernel load utilization
6. netbiosns_in: netbios name lookups (in)
7. netbiosns_out: netbios name lookups (out)
8. netbiosdgm_in: netbios name datagrams (in)
9. netbiosdgm_out: netbios name datagrams (out)
10. netbiosssn_in: netbios name sessions (in)
11. netbiosssn_out: netbios name sessions (out)
12. irc_in: IRC connections (in)
13. irc_out: IRC connections (out)
14. cfengine_in: CFEngine connections (in)
15. cfengine_out: CFEngine connections (out)
16. nfsd_in: nfs connections (in)
17. nfsd_out: nfs connections (out)
18. smtp_in: smtp connections (in)
19. smtp_out: smtp connections (out)
20. www_in: www connections (in)
21. www_out: www connections (out)
22. ftp_in: ftp connections (in)
23. ftp_out: ftp connections (out)
24. ssh_in: ssh connections (in)

25. ssh_out: ssh connections (out)
26. wwws_in: wwws connections (in)
27. wwws_out: wwws connections (out)
28. icmp_in: ICMP packets (in)
29. icmp_out: ICMP packets (out)
30. udp_in: UDP dgrams (in)
31. udp_out: UDP dgrams (out)
32. dns_in: DNS requests (in)
33. dns_out: DNS requests (out)
34. tcpsyn_in: TCP sessions (in)
35. tcpsyn_out: TCP sessions (out)
36. tcpack_in: TCP acks (in)
37. tcpack_out: TCP acks (out)
38. tcpfin_in: TCP finish (in)
39. tcpfin_out: TCP finish (out)
40. tcpmisc_in: TCP misc (in)
41. tcpmisc_out: TCP misc (out)
42. webaccess: Webserver hits
43. weberrors: Webserver errors
44. syslog: New log entries (Syslog)
45. messages: New log entries (messages)
46. temp0: CPU Temperature core 0
47. temp1: CPU Temperature core 1
48. temp2: CPU Temperature core 2
49. temp3: CPU Temperature core 3
50. cpu: %CPU utilization (all)
51. cpu0: %CPU utilization core 0
52. cpu1: %CPU utilization core 1
53. cpu2: %CPU utilization core 2
54. cpu3: %CPU utilization core 3

Slots with a higher number are used for custom measurement promises in CFEngine Nova.

These values collected and analysed by `cf-monitord` are transformed into agent variables in the `$(mon.`*`name`*`)` context.

Measurement promise syntax:

10.1.1 `stream_type`

Type: (menu option)

Allowed input range:

```
pipe
file
```

Synopsis: The datatype being collected.
Example:

```
stream_type => "pipe";
```

Notes:

CFEngine treats all input using a stream abstraction. The preferred interface is files, since they can be read without incurring the cost of a process. However pipes from executed commands may also be invoked.

10.1.2 data_type

Type: (menu option)
Allowed input range:

```
counter
int
real
string
slist
```

Synopsis: The datatype being collected.
Example:

```
  "/bin/df"

      handle => "free_disk_watch",
      stream_type => "pipe",

      data_type => "slist",

      history_type => "static",
      units => "device",
      match_value => file_systems,
      action => sample_min(10,15);
```

Notes:

When CFEngine (Nova) observes data, such as the attached partitions in the example above, the datatype determines how that data will be handled. Integer and real values, counters etc., are recorded as time-series if the history type is 'weekly', or as single values otherwise. If multiple items are matched by an observation, e.g. several lines in a file match the given regular expression, then these can be made into a list by choosing `slist`, else the first matching item will be selected.

10.1.3 history_type

Type: (menu option)
Allowed input range:

```
weekly
scalar
static
log
```

Synopsis: Whether the data can be seen as a time-series or just an isolated value
Example:

```
"/proc/meminfo"

     handle => "free_memory_watch",
     stream_type => "file",
     data_type => "int",
     history_type => "weekly",
     units => "kB",
     match_value => free_memory;
```

Notes:

'`scalar`' A single value, with compressed statistics is retained. The value of the data is not expected to change much for the lifetime of the daemon (and so will be sampled less often by '`cf-monitord`').

'`static`' A synonym for 'scalar'.

'`log`' The measured value is logged as an infinite time-series in '`$(sys.workdir)/state`'.

'`weekly`' A standard CFEngine two-dimensional time average (over a weekly period) is retained.

10.1.4 units

Type: string
Allowed input range: (arbitrary string)
Synopsis: The engineering dimensions of this value or a note about its intent used in plots
Example:

```
"/var/cfengine/state/cf_rootprocs"

   handle => "monitor_self_watch",
   stream_type => "file",
   data_type => "int",
   history_type => "weekly",
   units => "kB",
   match_value => proc_value(".*cf-monitord.*",

      "root\s+[0-9.]+\s+[0-9.]+\s+[0-9.]+\s+[0-9.]+\s+([0-9]+).*");
```

Notes:

This is an arbitary string used in documentation only.

10.1.5 match_value (body template)

Type: (ext body)

'select_line_matching'

Type: string

Allowed input range: .*

Synopsis: Regular expression for matching line location

Example:

```
# Editing

body location example
{
select_line_matching => "Expression match.* whole line";
}

# Measurement promises

body match_value example
{
select_line_matching => "Expression match.* whole line";
}
```

Notes:

The expression must match a whole line, not a fragment within a line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

This attribute is mutually exclusive of `select_line_number`.

'`select_line_number`'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Read from the n-th line of the output (fixed format)

Example:

```
body match_value find_line
{
select_line_number => "2";
}
```

Notes:

This is mutually exclusive of `select_line_matching`.

'`extraction_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression that should contain a single backreference for extracting a value

Example:

```
body match_value free_memory
{
select_line_matching => "MemFree:.*";
extraction_regex => "MemFree:\s+([0-9]+).*";
}
```

Notes:

A single parenthesized backreference should be given to lift the value to be measured out of the text stream. The regular expression may match a partial string (that is, it is unanchored see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'`track_growing_file`'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: If true, cfengine remembers the position to which is last read when opening the file, and resets to the start if the file has since been truncated

Example:

```
bundle monitor watch
{
measurements:

   "/home/mark/tmp/file"

         handle => "line_counter",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log("MYLINE.*"),
   history_type => "log",
         action => sample_rate("0");

}

#

body match_value scan_log(x)
{
select_line_matching => "^$(x)$";
track_growing_file => "true";
}

#

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

Notes:

This option applies only to file based input streams. If this is '`true`', CFEngine treats the file as if it were a log file, growing continuously. Thus the monitor reads all new entries since the last sampling time on each invocation. In this way, the monitor does not count lines in the log file redundantly.

This makes a log pattern promise equivalent to something like '`tail -f logfile | grep pattern`' in Unix parlance.

'`select_multiline_policy`'

Type: (menu option)

Allowed input range:

```
average
sum
first
last
```

Synopsis: Regular expression for matching line location

Example:

```
body match_value myvalue(xxx)
{
 select_line_matching => ".*$(xxx).*";
 extraction_regex => "root\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\s+(\S+).*";
 select_multiline_policy => "sum";
}
```

Notes:

History: Was introduced in 3.4.0 (2012)

This option governs how CFEngine handels multiple matching lines in the input stream. We can average or sum values if they are integer or real, or use first or last representative samples. If non-numerical data types are used on the the first match is used.

11 Special functions

11.1 Introduction to functions

There is a large number of functions built into CFEngine, and finding the right one to use can be a daunting task. The following tables are designed to make it easier for you to find the function you need, based on the value or type that the function returns or processes as inputs.

11.1.1 Functions listed by return value

Functions which return class

Section 11.2 [accessedbefore], page 415	Section 11.5 [and], page 419	Section 11.8 [changedbefore], page 421	Section 11.9 [classify], page 422	Section 11.10 [classmatch], page 422
Section 11.7 [concat], page 420	Section 11.17 [fileexists], page 427	Section 11.18 [filesexist], page 428	Section 11.28 [groupexists], page 437	Section 11.30 [hashmatch], page 439
Section 11.33 [hostinnetgroup], page 441	Section 11.34 [hostrange], page 441	Section 11.38 [iprange], page 445	Section 11.40 [isdir], page 446	Section 11.41 [isexecutable], page 447
Section 11.42 [isgreaterthan], page 447	Section 11.43 [islessthan], page 448	Section 11.44 [islink], page 449	Section 11.45 [isnewerthan], page 450	Section 11.46 [isplain], page 451
Section 11.47 [isvariable], page 452	Section 11.51 [ldaparray], page 455	Section 11.59 [or], page 461	Section 11.56 [not], page 460	Section 11.78 [regarray], page 483
Section 11.79 [regcmp], page 484	Section 11.80 [regextract], page 486	Section 11.84 [regldap], page 490	Section 11.82 [regline], page 488	Section 11.83 [reglist], page 489
Section 11.86 [remoteclassesmatching], page 493	Section 11.87 [returnszero], page 493	Section 11.90 [splayclass], page 497	Section 11.92 [strcmp], page 499	Section 11.95 [usemodule], page 502
Section 11.96 [userexists], page 503				

Functions which return (i,r,s)list

Section 11.23 [getindices], page 433	Section 11.25 [getusers], page 435	Section 11.27 [grep], page 436	Section 11.52 [ldaplist], page 456	Section 11.55 [maplist], page 459
Section 11.66 [peerleaders], page 469	Section 11.64 [peers], page 466	Section 11.71 [readintlist], page 474	Section 11.73 [readreallist], page 476	Section 11.76 [readstringlist], page 480

Section 11.91 [splitstring], page 498

Functions which return int

Section 11.3 [accumulated], page 416 Section 11.4 [ago], page 418 Section 11.12 [countlinesmatching], page 424 Section 11.14 [diskfree], page 425 Section 11.21 [getfields], page 431
Section 11.22 [getgid], page 432 Section 11.24 [getuid], page 434 Section 11.57 [now], page 460 Section 11.58 [on], page 460 Section 11.68 [randomint], page 472
Section 11.70 [readintarray], page 473 Section 11.72 [readrealarray], page 475 Section 11.74 [readstringarray], page 477 Section 11.75 [readstringarrayidx], page 479 Section 11.89 [selectservers], page 495

Functions which return (i,r)range

Section 11.39 [irange], page 446 Section 11.88 [rrange], page 494

Functions which return real

Section 11.67 [product], page 471 Section 11.93 [sum], page 500

Functions which return string

Section 11.6 [canonify], page 420 Section 11.15 [escape], page 426 Section 11.16 [execresult], page 426 Section 11.20 [getenv], page 430 Section 11.29 [hash], page 438
Section 11.31 [host2ip], page 440 Section 11.35 [hostsseen], page 442 Section 11.48 [join], page 453 Section 11.49 [lastnode], page 454 Section 11.53 [ldapvalue], page 457
Section 11.65 [peerleader], page 468 Section 11.69 [readfile], page 472 Section 11.77 [readtcp], page 482 Section 11.81 [registryvalue], page 487 Section 11.85 [remotescalar], page 491
Section 11.94 [translatepath], page 501

11.1.2 Functions which fill arrays

The following functions all fill arrays, although they return values which depend on the number of items processed.

Section 11.21 [getfields], page 431 Section 11.70 [readintarray], page 473 Section 11.72 [readrealarray], page 475 Section 11.74 [readstringarray], page 477 Section 11.75 [readstringarrayidx], page 479

Section 11.80 [regextract], page 486

11.1.3 Functions which read "large" data

The following functions read data from inside CFEngine (from classes, lists, strings, etc.) and outside of CFEngine (from files, databases, arrays, etc.)

Functions which read arrays

Section 11.23 [getindices], page 433
Section 11.26 [getvalues], page 435
Section 11.78 [regarray], page 483

Functions which read disk data

Section 11.14 [diskfree], page 425

Functions which read from a remote-CFEngine

Section 11.86 [remoteclassesmatching], page 493
Section 11.85 [remotescalar], page 491

Functions which read classes

Section 11.5 [and], page 419
Section 11.9 [classify], page 422
Section 11.10 [classmatch], page 422
Section 11.7 [concat], page 420
Section 11.56 [not], page 460
Section 11.59 [or], page 461

Functions which read command output

Section 11.16 [execresult], page 426
Section 11.87 [returnszero], page 493
Section 11.95 [usemodule], page 502

Functions which read the environment

Section 11.20 [getenv], page 430

Functions which read files

Section 11.12 [countlinesmatching], page 424
Section 11.21 [getfields], page 431
Section 11.25 [getusers], page 435
Section 11.30 [hashmatch], page 439
Section 11.65 [peerleader], page 468

Section 11.66 [peerleaders], page 469
Section 11.64 [peers], page 466
Section 11.69 [readfile], page 472
Section 11.70 [readintarray], page 473
Section 11.71 [readintlist], page 474
Section 11.72 [readrealarray], page 475
Section 11.73 [readreallist], page 476
Section 11.74 [readstringarray], page 477
Section 11.75 [readstringarrayidx], page 479
Section 11.76 [readstringlist], page 480
Section 11.82 [regline], page 488

Functions which read LDAP data

Section 11.51 [ldaparray], page 455
Section 11.52 [ldaplist], page 456
Section 11.53 [ldapvalue], page 457
Section 11.84 [regldap], page 490

Functions which read from the network

Section 11.77 [readtcp], page 482
Section 11.89 [selectservers], page 495

Functions which read the Windows registry

Section 11.81 [registryvalue], page 487

Functions which read (i,r,s)lists

Section 11.27 [grep], page 436
Section 11.48 [join], page 453
Section 11.67 [product], page 471
Section 11.83 [reglist], page 489
Section 11.93 [sum], page 500

Functions which read strings

Section 11.29 [hash], page 438
Section 11.49 [lastnode], page 454
Section 11.79 [regcmp], page 484
Section 11.80 [regextract], page 486
Section 11.91 [splitstring], page 498
Section 11.92 [strcmp], page 499
Section 11.94 [translatepath], page 501

11.1.4 Functions which look at file metadata

The following functions examine file metadata, but don't use the contents of the file.

Section 11.2 [accessedbefore], page 415
Section 11.8 [changedbefore], page 421
Section 11.17 [fileexists], page 427
Section 11.18 [filesexist], page 428
Section 11.40 [isdir], page 446

Section 11.44 [islink], page 449
Section 11.45 [isnewerthan], page 450
Section 11.46 [isplain], page 451

11.1.5 Functions which look at variables

Section 11.42 [isgreaterthan], page 447
Section 11.43 [islessthan], page 448
Section 11.47 [isvariable], page 452

11.1.6 Functions involving date or time

The following functions all do date or time computation

Section 11.2 [accessedbefore], page 415
Section 11.3 [accumulated], page 416
Section 11.4 [ago], page 418
Section 11.8 [changedbefore], page 421
Section 11.45 [isnewerthan], page 450
Section 11.57 [now], page 460
Section 11.58 [on], page 460
Section 11.90 [splayclass], page 497

11.1.7 Functions which work with or on regular expressions

Section 11.10 [classmatch], page 422
Section 11.12 [countlinesmatching], page 424
Section 11.15 [escape], page 426
Section 11.21 [getfields], page 431
Section 11.27 [grep], page 436
Section 11.70 [readintarray], page 473
Section 11.71 [readintlist], page 474
Section 11.72 [readrealarray], page 475
Section 11.73 [readreallist], page 476
Section 11.74 [readstringarray], page 477
Section 11.75 [readstringarrayidx], page 479
Section 11.76 [readstringlist], page 480
Section 11.78 [regarray], page 483
Section 11.79 [regcmp], page 484
Section 11.80 [regextract], page 486
Section 11.84 [regldap], page 490
Section 11.82 [regline], page 488
Section 11.83 [reglist], page 489
Section 11.91 [splitstring], page 498

11.2 Function accessedbefore

Synopsis: accessedbefore(arg1,arg2) returns type **class**

arg1 : Newer filename, *in the range* "?(/.*)
arg2 : Older filename, *in the range* "?(/.*)

True if arg1 was accessed before arg2 (atime)

Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { accessedbefore("/tmp/earlier","/tmp/later"), "linux" };

reports:

  do_it::

    "The secret changes have been accessed after the reference time";

}
```

Notes:

The function accesses the `atime` fields of a file and makes a comparison.

```
touch /tmp/reference
touch /tmp/secretfile

/var/cfengine/bin/cf-agent -f ./unit_accessed_before.cf -K
R: The secret changes have been accessed after the reference time
```

11.3 Function accumulated

Synopsis: accumulated(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Years, *in the range* 0,1000
arg2 : Months, *in the range* 0,1000
arg3 : Days, *in the range* 0,1000
arg4 : Hours, *in the range* 0,1000
arg5 : Minutes, *in the range* 0,1000
arg6 : Seconds, *in the range* 0,40000

Convert an accumulated amount of time into a system representation
Example:

```
bundle agent testbundle

{
processes:

 ".*"

    process_count   => anyprocs,
    process_select  => proc_finder;

reports:

 any_procs::

   "Found processes in range";
}

########################################################

body process_select proc_finder

{
ttime_range => irange(accumulated(0,0,0,0,2,0),accumulated(0,0,0,0,20,0));
process_result => "ttime";
}

########################################################

body process_count anyprocs

{
match_range => "0,0";
out_of_range_define => { "any_procs" };
}
```

Notes:

In the example we look for processes that have accumulated between 2 and 20 minutes of total run time.

ARGUMENTS:

The `accumulated` function measures total accumulated runtime. Arguments are applied additively, so that accumulated(0,0,2,27,90,0) means "2 days, 27 hours and 90 minutes of runtime" " – however, you are strongly encouraged to keep your usage of `accumulated` sensible and readable, e.g., accumulated(0,0,0,48,0,0) or accumulated(0,0,0,0,90,0).

'Years' Years of run time. For convenience in conversion, a year of runtime is always 365 days (one year equals 31,536,000 seconds).

'Month' Months of run time. For convenience in conversion, a month of runtime is always equal to 30 days of runtime (one month equals 2,592,000 seconds).

'Day' Days of runtime (one day equals 86,400 seconds)

'Hours' Hours of runtime

'Minutes' Minutes of runtime 0-59

'Seconds' Seconds of runtime

11.4 Function ago

Synopsis: ago(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Years, *in the range* 0,1000
arg2 : Months, *in the range* 0,1000
arg3 : Days, *in the range* 0,1000
arg4 : Hours, *in the range* 0,1000
arg5 : Minutes, *in the range* 0,1000
arg6 : Seconds, *in the range* 0,40000

Convert a time relative to now to an integer system representation
Example:

```
bundle agent testbundle

{
processes:

 ".*"

    process_count   => anyprocs,
    process_select  => proc_finder;

reports:

 any_procs::

   "Found processes out of range";
}

#########################################################

body process_select proc_finder
```

```
{
# Processes started between 5.5 hours and 20 minutes ago
stime_range => irange(ago(0,0,0,5,30,0),ago(0,0,0,0,20,0));
process_result => "stime";
}

########################################################

body process_count anyprocs

{
match_range => "0,0";
out_of_range_define => { "any_procs" };
}
```

Notes:

The ago function measures time relative to now. Arguments are applied in order, so that ago(0,18,55,27,0,0) means "18 months, 55 days, and 27 hours ago" – however, you are strongly encouraged to keep your usage of ago sensible and readable, e.g., ago(0,0,120,0,0,0) or ago(0,0,0,72,0,0).

ARGUMENTS:

'Years' Years ago. If today is February 29, and "**n** years ago" is not within a leap-year, February 28 will be used.

'Month' Months ago. If the current month has more days that "**n** months ago", the last day of "**n** months ago" will be used (e.g., if today is April 31 and you compute a date 1 month ago, the resulting date will be March 30). equal to 30 days of runtime (one month equals 2,592,000 seconds).

'Day' Days ago (you may, for example, specify 120 days)

'Hours' Hours ago. Since all computation are done using "Epoch time", 1 hour ago will alway result in a time 60 minutes in the past, even during the transition from Da ylight time to Standard time.

'Minutes' Minutes ago 0-59

'Seconds' Seconds ago

11.5 Function and

Synopsis: and(...) returns type **string**

Calculate whether all arguments evaluate to true

Example:

```
commands:
  "/usr/bin/generate_config $(config)"
    ifvarclass => and(not(fileexists("/etc/config/$(config)")), "generating_configs");
```

Notes:

History: Was introduced in 3.2.0, Nova 2.1.0 (2011)

11.6 Function canonify

Synopsis: canonify(arg1) returns type **string**

arg1 : String containing non-identifier characters, *in the range* .*

Convert an abitrary string into a legal class name

Example:

```
commands:

   "/var/cfengine/bin/$(component)"

       ifvarclass => canonify("start_$(component)");
```

Notes:

This is for use in turning arbitrary text into class data, See Section 11.9 [Function classify], page 422.

11.7 Function concat

Synopsis: concat(...) returns type **string**

Concatenate all arguments into string

Example:

```
commands:
  "/usr/bin/generate_config $(config)"
```

```
    ifvarclass => concat("have_config_", canonify("$(config)"));
```

Notes:

History: Was introduced in 3.2.0, Nova 2.1.0 (2011)

11.8 Function changedbefore

Synopsis: changedbefore(arg1,arg2) returns type **class**

arg1 : Newer filename, *in the range* "?(/.*)
arg2 : Older filename, *in the range* "?(/.*)

True if arg1 was changed before arg2 (ctime)

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { changedbefore("/tmp/earlier","/tmp/later"), "linux" };

reports:

  do_it::

    "The derived file needs updating";

}
```

Notes:

Change times include both file permissions and file contents. Comparisons like this are normally used for updating files (like the 'make' command).

11.9 Function classify

Synopsis: classify(arg1) returns type **class**

arg1 : Input string, *in the range* .*

True if the canonicalization of the argument is a currently defined class
Example:

```
classes:

 "i_am_the_policy_host" expression => classify("master.example.org");
```

Notes:

This function returns true if the canonical form of the argument is already a defined class. This is useful, for example, transforming variables into classes, See Section 11.6 [Function canonify], page 420.

11.10 Function classmatch

Synopsis: classmatch(arg1) returns type **class**

arg1 : Regular expression, *in the range* .*

True if the regular expression matches any currently defined class
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { classmatch(".*_cfengine_com"), "linux" };
```

```
reports:

  do_it::

    "Host matches pattern";

}
```

Notes:

The regular expression is matched against the current list of defined classes. The regular expression must match a complete class for the expression to be true, that is, the regex is anchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

11.11 Function countclassesmatching

Synopsis: countclassesmatching(arg1) returns type **int**

arg1 : Regular expression, *in the range* .*

Count the number of defined classes matching regex arg1

Example:

```
bundle agent example
{
vars:

  "num" int => countclassesmatching("entropy.*low");

reports:

  cfengine_3::

    "Found $(num) classes matching";

}
```

Notes:

This function matches classes, using a regular expression that should match the whole line.

'regex' A regular expression matching zero or more classes in the current list of defined classes. The regular expression must match a complete class, that is, it is anchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

The function returns the number of classes matched.

11.12 Function countlinesmatching

Synopsis: countlinesmatching(arg1,arg2) returns type **int**

arg1 : Regular expression, *in the range* .*
arg2 : Filename, *in the range* "?(/.*)

Count the number of lines matching regex arg1 in file arg2

Example:

```
bundle agent example
{
vars:

  "no" int => countlinesmatching("m.*","/etc/passwd");

reports:

  cfengine_3::

    "Found $(no) lines matching";

}
```

Notes:

This function matches lines in the named file, using a regular expression that should match the whole line.

'regex' A regular expression matching zero or more lines. The regular expression must match a complete line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'filename' The name of the file to be examined.

The function returns the number of lines matched.

11.13 Function dirname

Synopsis: dirname(arg1) returns type **string**

arg1 : File path, *in the range* .*

Return the parent directory name for given path

Example:

```
vars:
  "apache_dir" string => dirname("/etc/apache2/httpd.conf");
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

This function returns directory name for the argument. If directory name is provided, name of parent directory is returned.

11.14 Function diskfree

Synopsis: diskfree(arg1) returns type **int**

arg1 : File system directory, *in the range* "?(/.*)

Return the free space (in KB) available on the directory's current partition (0 if not found)

Example:

```
bundle agent example
{
vars:

  "free" int => diskfree("/tmp");

reports:

  cfengine_3::

    "Freedisk $(free)";

}
```

Notes:

Values returned in kilobytes.

11.15 Function escape

Synopsis: escape(arg1) returns type **string**

arg1 : IP address or string to escape, *in the range* .*

Escape regular expression characters in a string

Example:

```
bundle server control
{
allowconnects         => { "127\.0\.0\.1", escape("192.168.2.1") };
}
```

Notes:

This function is useful for making inputs readable when a regular expression is required, but the literal string contains special characters. The function simply 'escapes' all the regular expression characters, so that you don't have to.

This in the example above, the string "192.168.2.1" is "escaped" to be equivalent to "192\.168\.2\.1" (because without the backslashes, the regular expression "192.168.2.1" will also match the IP ranges "192.168.201", "192.168.231", etc - since the dot character means "match any character" when used in a regular expression).

History: This function was introduced in CFEngine version 3.0.4 (2010)

11.16 Function execresult

Synopsis: execresult(arg1,arg2) returns type **string**

arg1 : Fully qualified command path, *in the range* "?(/.*)
arg2 : Shell encapsulation option, *in the range* useshell,noshell

Execute named command and assign output to variable

Example:

```
body common control

{
bundlesequence  => { "example" };
}
```

```
#########################################################

bundle agent example

{
vars:

  "my_result" string => execresult("/bin/ls /tmp","noshell");

reports:

  linux::

    "Variable is $(my_result)";

}
```

Notes:

The second argument (‘`useshell`’/‘`noshell`’) decides whether a shell will be used to encapsulate the command. This is necessary in order to combine commands with pipes etc, but remember that each command requires a new process that reads in files beyond CFEngine’s control. Thus using a shell is both a performance hog and a potential security issue.

Note: you should never use this function to execute comands that make changes to the system, or perform lengthy computations. Such an operation is beyond CFEngine’s ability to guarantee convergence, and on multiple passes and during syntax verification, these function calls are executed resulting in system changes that are ‘covert’. Calls to `execresult` should be for discovery and information extraction only.

Note: if the command is not found, the result will be the empty string!

Change: policy change in CFEngine 3.0.5. Previously newlines were changed for spaces, now newlines are preserved.

11.17 Function fileexists

Synopsis: fileexists(arg1) returns type **class**

arg1 : File object name, *in the range* "?(/.*)

True if the named file can be accessed

Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "exists" expression => fileexists("/etc/passwd");

reports:

  exists::

    "File exists";

}
```

Notes:

The user must have access permissions to the file for this to work faithfully.

11.18 Function filesexist

Synopsis: filesexist(arg1) returns type **class**

arg1 : Array identifier containing list, *in the range* @[(][a-zA-Z0-9]+[)]

True if the named list of files can ALL be accessed
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example
```

```
{
vars:

  "mylist" slist => { "/tmp/a", "/tmp/b", "/tmp/c" };

classes:

  "exists" expression => filesexist("@(mylist)");

reports:

  exists::

    "Files exist";

  !exists::

    "Do not exist";

}
```

Notes:

The user must have access permissions to the file for this to work faithfully.

11.19 Function filesize

Synopsis: filesize(arg1) returns type **int**

arg1 : File object name, *in the range* "?(/.*)

Returns the size in bytes of the file
Example:

```
bundle agent example
{
vars:

  "exists" int => filesize("/etc/passwd");
  "nexists" int => filesize("/etc/passwdx");

reports:
```

```
  !xyz::

    "File size $(exists)";
    "Does not exist $(nexists)";

}
```

Notes:

History: Was introduced in version 3.1.3,Nova 2.0.2 (2010)

If the file object does not exist, the function call fails and the variable does not expand.

11.20 Function getenv

Synopsis: getenv(arg1,arg2) returns type **string**

arg1 : Name of environment variable, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : Maximum number of characters to read , *in the range* 0,99999999999

Return the environment variable named arg1, truncated at arg2 characters

Example:

```
bundle agent example
{
vars:

   "myvar" string => getenv("PATH","20");

classes:

  "isdefined" not => strcmp("$(myvar)","");

reports:

  isdefined::

   "The path is $(myvar)";

  !isdefined::

   "The named variable PATH does not exist";
```

```
}
```

Notes:

Returns an empty string if the environment variable is not defined. Arg2 is used to avoid unexpectedly large return values, which could lead to security issues. Choose a reasonable value based on the environment variable you are querying.

History: This function was introduced in CFEngine version 3.0.4 (2010)

11.21 Function getfields

Synopsis: getfields(arg1,arg2,arg3,arg4) returns type **int**

arg1 : Regular expression to match line, *in the range* .*
arg2 : Filename to read, *in the range* "?(/.*)
arg3 : Regular expression to split fields, *in the range* .*
arg4 : Return array name, *in the range* .*

Get an array of fields in the lines matching regex arg1 in file arg2, split on regex arg3 as array name arg4

Example:

```
bundle agent example

{
vars:

  "no" int => getfields("mark:.*","/etc/passwd",":","userdata");

reports:

  cfengine_3::

    "Found $(no) lines matching";
    "Mark's homedir = $(userdata[6])";

}
```

Notes:

This function matches lines (using a regular expression) in the named file, and splits the *first* matched line into fields (using a second) regular expression), placing these into a named array whose

elements are `array[1],array[2],...` This is useful for examining user data in the Unix password or group files.

'`regex`' A regular expression matching one or more lines. The regular expression must match the entire line (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'`filename`' The name of the file to be examined.

'`split`' A regex pattern which is used to parse the field separator(s) to split up the file into items

'`array_lval`'
The base name of the array that returns the values.

The function returns the number of lines matched. This function is most useful when you want only the first matching line (e.g., to mimic the behavior of the *getpwnam(3)* on the file '`/etc/passwd`'). If you want to examine *all* matching lines, See Section 11.74 [Function readstringarray], page 477, instead.

11.22 Function getgid

Synopsis: getgid(arg1) returns type **int**

arg1 : Group name in text, *in the range* .*

Return the integer group id of the named group on this host

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "gid" int => getgid("users");

reports:

  Yr2008::

    "Users gid is $(gid)";
```

```
}
```

Notes:

If the named group does not exist, the variable will not be defined. On windows, which does not support group ids, the variable will not be defined.

11.23 Function getindices

Synopsis: getindices(arg1) returns type **slist**

arg1 : Cfengine array identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Get a list of keys to the array whose id is the argument and assign to variable

Example:

```
body common control

{
any::

  bundlesequence  => { "testsetvar" };
}


#######################################################

bundle agent testsetvar

{
vars:

  "v[index_1]" string => "value_1";
  "v[index_2]" string => "value_2";

  "parameter_name" slist => getindices("v");

reports:

  Yr2008::

   "Found index: $(parameter_name)";
```

```
}
```

Notes:

Make sure you specify the correct scope when supplying the name of the variable.

11.24 Function getuid

Synopsis: getuid(arg1) returns type **int**

arg1 : User name in text, *in the range* .*

Return the integer user id of the named user on this host

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "uid" int => getuid("mark");

reports:

  Yr2008::

    "Users uid is $(uid)";

}
```

Notes:

If the named user is not registered the variable will not be defined. On windows, which does not support user ids, the variable will not be defined.

11.25 Function getusers

Synopsis: getusers(arg1,arg2) returns type **slist**

arg1 : Comma separated list of User names, *in the range* .*
arg2 : Comma separated list of UserID numbers, *in the range* .*

Get a list of all system users defined, minus those names defined in arg1 and uids in arg2
Example:

```
vars:
  "allusers" slist => getusers("zenoss,mysql,at","12,0");

reports:

 linux::

  "Found user $(allusers)";
```

Notes:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010) This function is only available on Unix-like systems in the present version.

The function has two arguments, both are comma separated lists. The first argument is a list of user names that should be excluded from the output. The second is a list of integrer UIDs that should be excluded.

11.26 Function getvalues

Synopsis: getvalues(arg1) returns type **slist**

arg1 : Cfengine array identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Get a list of values corresponding to the right hand sides in an array whose id is the argument and assign to variable
Example:

```
body common control

{
any::

  bundlesequence  => { "testsetvar" };
}
```

```
#######################################################

bundle agent testsetvar

{
vars:

  "v[index_1]" string => "value_1";
  "v[index_2]" string => "value_2";

  "parameter_name" slist => getvalues("v");

reports:

  Yr2008::

   "Found index: $(parameter_name)";

}
```

Notes:

Make sure you specify the correct scope when supplying the name of the variable. If the array contains list elements on the right hand side then all of the list elements are flattened into a single list to make the return value a list.

11.27 Function grep

Synopsis: grep(arg1,arg2) returns type **slist**

arg1 : Regular expression, *in the range* .*
arg2 : CFEngine list identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Extract the sub-list if items matching the regular expression in arg1 of the list named in arg2
Example:

```
bundle agent test

{
vars:

  "mylist" slist => { "One", "Two", "Three", "Four", "Five" };
```

```
  "sublist" slist => grep("T.*","mylist");

  "empty_list" slist => grep("ive","mylist");

reports:

 linux::

  "Item: $(sublist)";

}
```

Notes:

Extracts a sublist of elements matching the regular expression in arg1 from a list variable specified in arg2. The regex is anchored. See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

11.28 Function groupexists

Synopsis: groupexists(arg1) returns type **class**

arg1 : Group name or identifier, *in the range* .*

True if group or numerical id exists on this host

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "gname" expression => groupexists("users");
  "gid"   expression => groupexists("100");
```

```
reports:

  gname::

    "Group exists by name";

  gid::

    "Group exists by id";

}
```

Notes:

The group may be specified by name or number.

11.29 Function hash

Synopsis: hash(arg1,arg2) returns type **string**

arg1 : Input text, *in the range* .*
arg2 : Hash or digest algorithm, *in the range* md5,sha1,sha256,sha512,sha384,crypt

Return the hash of arg1, type arg2 and assign to a variable
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "md5" string => hash("CFEngine is not cryptic","md5");

reports:
```

```
  Yr2008::

    "Hashed to: $(md5)";

}
```

Notes:

Hash functions are extremely sensitive to input. You should not expect to get the same answer from this function as you would from every other tool, since it depends on how whitespace and end of file characters are handled.

Valid hash types (depending on availablity) include: `md5`, `sha1`, `sha256`, `sha512` ,`sha384`, `crypt`.

11.30 Function hashmatch

Synopsis: hashmatch(arg1,arg2,arg3) returns type **class**

arg1 : Filename to hash, *in the range* "?(/.*)
arg2 : Hash or digest algorithm, *in the range* md5,sha1,crypt,cf_sha224,cf_sha256,cf_sha384,cf_sha512

arg3 : ASCII representation of hash for comparison, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Compute the hash of arg1, of type arg2 and test if it matches the value in arg3
Example:

```
bundle agent example

{
classes:

  "matches" expression => hashmatch("/etc/passwd","md5","c5068b7c2b1707f8939b283a2758a691");

reports:

  matches::

    "File has correct version";

}
```

Notes:

```
(class) hashmatch(file,md5|sha1|crypt,hash-comparison);
```

This function may be used to determine whether a system has a particular version of a binary file (e.g. software patch).

ARGUMENTS:

- The file concerned
- The type of hash
- A string of the hash to which we expect the file to conform.

11.31 Function host2ip

Synopsis: host2ip(arg1) returns type **string**

arg1 : Host name in ascii, *in the range* .*

Returns the primary name-service IP address for the named host

Example:

```
bundle server control
{
allowconnects         => { escape(host2ip("www.example.com")) };
}
```

Notes:

Uses whatever configured name service is used by the resolver library to translate a hostname into an IP address. It will return an IPv6 address by preference if such an address exists. This function uses the standard lookup procedure for a name, so it mimics internal processes and can therefore be used not only to cache multiple lookups in the configuration, but to debug the behaviour of the resolver.

History: This function was introduced in CFEngine version 3.0.4 (2010)

11.32 Function ip2host

Synopsis: ip2host(arg1) returns type **string**

arg1 : IP address (IPv4 or IPv6), *in the range* .*

Returns the primary name-service host name for the IP address

Example:

```
bundle agent reverse_lookup
```

```
{
vars:
 "local4" string => ip2host("127.0.0.1");
 "local6" string => ip2host("::1");


reports:
cfengine_3::
  "local4 is $(local4)";
  "local6 is $(local6)";
}
```

Notes:

Uses whatever configured name service is used by the resolver library to translate an IP address to a hostname. IPv6 addresses will also resolve, if supported by the resolver library.

Note that DNS lookups may take time and thus cause CFEngine agents to wait for responses, slowing their progress significantly.

History: Was introduced in version 3.1.3, Nova 2.0.2 (2010)

11.33 Function hostinnetgroup

Synopsis: hostinnetgroup(arg1) returns type **class**

arg1 : Netgroup name, *in the range* .*

True if the current host is in the named netgroup

Example:

```
classes:

  "ingroup" expression => hostinnetgroup("my_net_group");
```

Notes:

11.34 Function hostrange

Synopsis: hostrange(arg1,arg2) returns type **class**

arg1 : Hostname prefix, *in the range* .*
arg2 : Enumerated range, *in the range* .*

True if the current host lies in the range of enumerated hostnames specified

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "compute_nodes" expression => hostrange("cpu-","01-32");

reports:

  compute_nodes::

    "No computer is a cluster";

}
```

Notes:

This is a pattern matching function for non-regular (enumerated) expressions.

11.35 Function hostsseen

Synopsis: hostsseen(arg1,arg2,arg3) returns type **slist**

arg1 : Horizon since last seen in hours, *in the range* 0,99999999999
arg2 : Complements for selection policy, *in the range* lastseen,notseen
arg3 : Type of return value desired, *in the range* name,address

Extract the list of hosts last seen/not seen within the last arg1 hours
Example:

```
bundle agent test
```

```
{
vars:

  "myhosts" slist => { hostsseen("inf","lastseen","address") };

reports:

  cfengine_3::

    "Found client/peer: $(myhosts)";

}
```

Notes:

Finds a list of hosts seen by a CFEngine remote connection on the current host within the number of hours specified by argument 1. Argument 2 may be '`lastseen`' or '`notseen`', the latter being all hosts not observed to have connected within the specified time. Argument 3 may be '`address`' or '`name`', to return ip address or hostname form.

11.36 Function hostswithclass

Synopsis: hostswithclass(arg1,arg2) returns type **slist**

arg1 : Class name to look for, *in the range* [a-zA-Z0-9_]+
arg2 : Type of return value desired, *in the range* name,address

Extract the list of hosts with the given class set from the hub database (commercial extension)
Example:

```
body common control
{
bundlesequence => { "test" };
inputs => { "cfengine_stdlib.cf" };
}


bundle agent test
{
vars:

am_policy_hub::
 "host_list" slist => hostswithclass( "debian", "name" );
```

```
files:
am_policy_hub::
  "/tmp/master_config.cfg"
         edit_line => insert_lines("host=$(host_list)"),
            create => "true";
}
```

Notes:

On CFEngine Nova hubs, this function can be used to return a list of hostnames or ip-addresses of hosts that has the class given as argument 1. Argument 2 may be '`address`' or '`name`', to return ip address or hostname form.

Note that this function only works locally on the hub, but allows the hub to construct custom configuration files for (classes of) hosts.

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

Availability: Enterprise editions of CFEngine only.

11.37 Function hubknowledge

Synopsis: hubknowledge(arg1) returns type **string**

arg1 : Variable identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Read global knowledge from the hub host by id (commercial extension)

Example:

```
vars:

  guard::

   "global_number" string => hubknowledge("number_variable");
```

Notes:

This function is only available in commercial releases of CFEngine. It is intended for use in distributed orchestration. It is recommended that you use this function sparingly with *guards*, as it contributes to network traffic and depends on the network for its function. Unlike `remotescalar()`, the value of hub-knowledge is not cached.

This function behaves is essentially similar to the `remotescalar` function, except that it always gets its information from the policy server hub by an encrypted connection. It is designed for spreading globally calibrated information about a CFEngine swarm back to the client machines. The data available

through this channel are generated automatically by discovery, unlike `remotescalar` which accesses user defined data.

This function asks for an identifier; it is up to the server to interpret what this means and to return a value of its choosing. If the identifier matches a persistent scalar variable (such as is used to count distributed processes in CFEngine Enterprise) then this will be returned preferentially. If no such variable is found, then the server will look for a literal string in a server bundle with a handle that matches the requested object.

11.38 Function iprange

Synopsis: iprange(arg1) returns type **class**

arg1 : IP address range syntax, *in the range* .*

True if the current host lies in the range of IP addresses specified

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "adhoc_group_1" expression => iprange("128.39.89.10-15");
  "adhoc_group_2" expression => iprange("128.39.74.1/23");

reports:

  adhoc_group_1::

    "Some numerology";

  adhoc_group_2::

    "The masked warriors";
}
```

Notes:

Pattern matching based on IP addresses.

11.39 Function irange

Synopsis: irange(arg1,arg2) returns type **irange [int,int]**

arg1 : Integer, *in the range* -99999999999,9999999999
arg2 : Integer, *in the range* -99999999999,9999999999

Define a range of integer values for cfengine internal use
Example:

```
irange("1","100");

irange(ago(0,0,0,1,30,0), "0");
```

Notes:

Used for any scalar attribute which requires an integer range. You can generally interchangeably say '`"1,10"`' or '`irange("1","10")`' (however, if you want to create a range of dates or times, you must use irange if you also use the functions '`ago`', '`now`', '`accumulated`', etc).

11.40 Function isdir

Synopsis: isdir(arg1) returns type **class**

arg1 : File object name, *in the range* "?(/.*)

True if the named object is a directory
Example:

```
body common control

{
bundlesequence  => { "example" };
}
```

```
#########################################################

bundle agent example

{
classes:

  "isdir" expression => isdir("/etc");

reports:

  isdir::

    "Directory exists..";

}
```

Notes:

The CFEngine process must have access to the object concerned in order for this to work.

11.41 Function isexecutable

Synopsis: isexecutable(arg1) returns type **class**

arg1 : File object name, *in the range* "?(/.*)

True if the named object has execution rights for the current user

Example:

```
classes:

  "yes" expression => isexecutable("/bin/ls");
```

Notes:

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

11.42 Function isgreaterthan

Synopsis: isgreaterthan(arg1,arg2) returns type **class**

arg1 : Larger string or value, *in the range* .*

arg2 : Smaller string or value, *in the range* .*

True if arg1 is numerically greater than arg2, else compare strings like strcmp
Example:

```
body common control

{
bundlesequence  => { "test"  };
}

###########################################################

bundle agent test

{
classes:

  "ok" expression => isgreaterthan("1","0");

reports:

  ok::

    "Assertion is true";

 !ok::

  "Assertion is false";

}
```

Notes:

The comparison is made numerically if possible. If the values are strings, the result is identical to that of comparing with 'strcmp()'.

11.43 Function islessthan

Synopsis: islessthan(arg1,arg2) returns type **class**

arg1 : Smaller string or value, *in the range* .*
arg2 : Larger string or value, *in the range* .*

True if arg1 is numerically less than arg2, else compare strings like NOT strcmp

Example:

```
body common control

{
bundlesequence  => { "test"  };
}

###########################################################

bundle agent test

{
classes:

  "ok" expression => islessthan("0","1");

reports:

  ok::

    "Assertion is true";

 !ok::

  "Assertion is false";

}
```

Notes:

The complement of `isgreaterthan`. The comparison is made numerically if possible. If the values are strings, the result is identical to that of comparing with '`strcmp()`'.

11.44 Function islink

Synopsis: islink(arg1) returns type **class**

arg1 : File object name, *in the range* "?(/.*)

True if the named object is a symbolic link

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "isdir" expression => islink("/tmp/link");

reports:

  isdir::

    "Directory exists..";

}
```

Notes:

The link node must both exist and be a symbolic link. Hard links cannot be detected using this function. A hard link is a regular file or directory.

11.45 Function isnewerthan

Synopsis: isnewerthan(arg1,arg2) returns type **class**

arg1 : Newer file name, *in the range* "?(/.*)
arg2 : Older file name, *in the range* "?(/.*)

True if arg1 is newer (modified later) than arg2 (mtime)
Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { isnewerthan("/tmp/later","/tmp/earlier"), "linux" };

reports:

  do_it::

    "The derived file needs updating";

}
```

Notes:

This function compares the modification time of the file, referring to changes of content only.

11.46 Function isplain

Synopsis: isplain(arg1) returns type **class**

arg1 : File object name, *in the range* "?(/.*)

True if the named object is a plain/regular file
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example
```

```
{
classes:

  "isplain" expression => isplain("/etc/passwd");

reports:

  isplain::

    "File exists..";

}
```

Notes:

11.47 Function isvariable

Synopsis: isvariable(arg1) returns type **class**

arg1 : Variable identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

True if the named variable is defined

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "bla" string => "xyz..";

classes:

  "exists" expression => isvariable("bla");
```

```
reports:

  exists::

    "Variable exists: \"$(bla)\"..";

}
```

Notes:

The variable need only exist. This says nothing about its value. Use `regcmp` to check variable values.

11.48 Function join

Synopsis: join(arg1,arg2) returns type **string**

arg1 : Join glue-string, *in the range* .*
arg2 : CFEngine list identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Join the items of arg2 into a string, using the conjunction in arg1
Example:

```
bundle agent test

{
vars:

  "mylist" slist => { "one", "two", "three", "four", "five" };

  "scalar" string => join("<->","mylist");

reports:

 linux::

  "Concatenated $(scalar)";

}
```

Notes:

Converts a string of type list into a scalar variable using the join string in first argument.

11.49 Function lastnode

Synopsis: lastnode(arg1,arg2) returns type **string**

arg1 : Input string, *in the range* .*
arg2 : Link separator, e.g. /,:, *in the range* .*

Extract the last of a separated string, e.g. filename from a path
Example:

```
bundle agent yes
{
vars:

  "path1" string => "/one/two/last1";
  "path2" string => "one:two:last2";

  "last1" string => lastnode("$(path1)","/");
  "last2" string => lastnode("$(path2)",":");

  "last3" string => lastnode("$(path2)","/");

reports:

  Yr2009::

    "Last = $(last1),$(last2),$(last3)";

}
```

Notes:

This function returns the final node in a chain, given a regular expression to split on. This is mainly useful for finding leaf-names of files, from a fully qualified path name.

11.50 Function laterthan

Synopsis: laterthan(arg1,arg2,arg3,arg4,arg5,arg6) returns type **class**

arg1 : Years, *in the range* 0,1000
arg2 : Months, *in the range* 0,1000
arg3 : Days, *in the range* 0,1000

arg4 : Hours, *in the range* 0,1000
arg5 : Minutes, *in the range* 0,1000
arg6 : Seconds, *in the range* 0,40000

True if the current time is later than the given date

Example:

```
classes:

  "after_deadline" expression => laterthan(2000,1,1,0,0,0);
```

Notes:

The arguments are standard time, See Section 11.58 [Function on], page 460.

11.51 Function ldaparray

Synopsis: ldaparray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **class**

arg1 : Array name, *in the range* .*
arg2 : URI, *in the range* .*
arg3 : Distinguished name, *in the range* .*
arg4 : Filter, *in the range* .*
arg5 : Search scope policy, *in the range* subtree,onelevel,base
arg6 : Security level, *in the range* none,ssl,sasl

Extract all values from an ldap record

Example:

```
classes:

   "gotdata" expression => ldaparray(
                                    "myarray",
                                    "ldap://ldap.example.org",
                                    "dc=cfengine,dc=com",
                                    "(uid=mark)",
                                    "subtree",
                                    "none");
```

Notes:

```
(class) ldaparray (array,uri,dn,filter,scope,security)
```

This function retrieves an entire record with all elements and populates an associative array with the entries. It returns a class which is true if there was a match for the search and false if nothing was retrieved.

ARGUMENTS:

'`array`' String name of the array to populate with the result of the search

'`uri`' String value of the ldap server. e.g. `"ldap://ldap.cfengine.com.no"`

'`dn`' Distinguished name, an ldap formatted name built from components, e.g. "dc=cfengine,dc=com".

'`filter`' String filter criterion, in ldap search, e.g. "(sn=User)".

'`scope`' Menu option, the type of ldap search, from

```
subtree
onelevel
base
```

'`security`' Menu option indicating the encryption and authentication settings for communication with the LDAP server. These features might be subject to machine and server capabilites.

```
none
ssl
sasl
```

11.52 Function ldaplist

Synopsis: ldaplist(arg1,arg2,arg3,arg4,arg5,arg6) returns type **slist**

arg1 : URI, *in the range* .*
arg2 : Distinguished name, *in the range* .*
arg3 : Filter, *in the range* .*
arg4 : Record name, *in the range* .*
arg5 : Search scope policy, *in the range* subtree,onelevel,base
arg6 : Security level, *in the range* none,ssl,sasl

Extract all named values from multiple ldap records

Example:

```
vars:

  # Get all matching values for "uid" - should be a single record match
```

```
  "list" slist =>  ldaplist(
                           "ldap://ldap.example.org",
                           "dc=cfengine,dc=com",
                           "(sn=User)",
                           "uid",
                           "subtree",
                           "none"
                           );
```

Notes:

```
(slist) ldaplist(uri,dn,filter,name,scope,security)
```

This function retrieves a single field from all matching LDAP records identified by the search parameters.

ARGUMENTS:

'`uri`' String value of the ldap server. e.g. "`ldap://ldap.cfengine.com.no`"

'`dn`' Distinguished name, an ldap formatted name built from components, e.g. "dc=cfengine,dc=com".

'`filter`' String filter criterion, in ldap search, e.g. "(sn=User)".

'`name`' String value, the name of a single record to be retrieved, e.g. `uid`.

'`scope`' Menu option, the type of ldap search, from the specified root. May take values:

```
subtree
onelevel
base
```

'`security`' Menu option indicating the encryption and authentication settings for communication with the LDAP server. These features might be subject to machine and server capabilites.

```
none
ssl
sasl
```

11.53 Function ldapvalue

Synopsis: ldapvalue(arg1,arg2,arg3,arg4,arg5,arg6) returns type **string**

arg1 : URI, *in the range* .*
arg2 : Distinguished name, *in the range* .*
arg3 : Filter, *in the range* .*
arg4 : Record name, *in the range* .*
arg5 : Search scope policy, *in the range* subtree,onelevel,base
arg6 : Security level, *in the range* none,ssl,sasl

Extract the first matching named value from ldap

Example:

```
vars:

   # Get the first matching value for "uid" in schema

  "value" string => ldapvalue(
                             "ldap://ldap.example.org",
                             "dc=cfengine,dc=com",
                             "(sn=User)",
                             "uid",
                             "subtree",
                             "none"
                             );
```

Notes:

```
(string) ldapvalue(uri,dn,filter,name,scope,security)
```

This function retrieves a single field from a single LDAP record identified by the search parameters. The first matching value it taken.

ARGUMENTS:

'`uri`' String value of the ldap server. e.g. "`ldap://ldap.cfengine.com.no`"

'`dn`' Distinguished name, an ldap formatted name built from components, e.g. "dc=cfengine,dc=com".

'`filter`' String filter criterion, in ldap search, e.g. "(sn=User)".

'`name`' String value, the name of a single record to be retrieved, e.g. `uid`.

'`scope`' Menu option, the type of ldap search, from the specified root. May take values:

```
subtree
onelevel
base
```

11.54 Function lsdir

Synopsis: lsdir(arg1,arg2,arg3) returns type **slist**

arg1 : Path to base directory, *in the range* .+
arg2 : Regular expression to match files or blank, *in the range* .*

arg3 : Include the base path in the list, *in the range* true,false,yes,no,on,off

Return a list of files in a directory matching a regular expression

Example:

```
vars:
  "listfiles" slist => lsdir("/etc", "(passwd|shadow).*", "true");
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

This function returns list of files in directory specified in arg1, matched with regular expression in arg2. In case arg3 is true, full paths are returned, otherwise only names relative to the the directory are returned.

11.55 Function maplist

Synopsis: maplist(arg1,arg2) returns type **slist**

arg1 : Pattern based on $(this) as original text, *in the range* .*
arg2 : The name of the list variable to map, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Return a list with each element modified by a pattern based $(this)

Example:

```
bundle agent test
{
vars:

  "oldlist" slist => { "a", "b", "c" };
  "newlist" slist => maplist("Element ($(this))","oldlist");

reports:
 linux::
  "Transform: $(newlist)";
}
```

Notes:

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

This is essentially like the map() function in Perl, and applies to lists.

11.56 Function not

Synopsis: not(arg1) returns type **string**

arg1 : Class value, *in the range* .*

Calculate whether argument is false

Example:

```
commands:
  "/usr/bin/generate_config $(config)"
    ifvarclass => not(fileexists("/etc/config/$(config)"));
```

Notes:

History: Was introduced in 3.2.0, Nova 2.1.0 (2011)

11.57 Function now

Synopsis: now() returns type **int**

Convert the current time into system representation

Example:

```
body file_select zero_age
{
mtime       => irange(ago(1,0,0,0,0,0),now);
file_result => "mtime";
}
```

Notes:

11.58 Function on

Synopsis: on(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Year, *in the range* 1970,3000
arg2 : Month, *in the range* 1,12
arg3 : Day, *in the range* 1,31
arg4 : Hour, *in the range* 0,23

arg5 : Minute, *in the range* 0,59
arg6 : Second, *in the range* 0,59

Convert an exact date/time to an integer system representation
Example:

```
body file_select zero_age
{
mtime       => irange(on(2000,1,1,0,0,0),now);
file_result => "mtime";
}
```

Notes:

An absolute date in the local timezone. Note that in process matching dates could be wrong by an hour depending on Daylight Savings Time / Summer Time. This is a known bug to be fixed.
ARGUMENTS:

‘Years’ The year, e.g. 2009

‘Month’ The Month, 1-12

‘Day’ The day 1-31

‘Hours’ The hour 0-23

‘Minutes’ The minutes 0-59

‘Seconds’ The number of seconds 0-59

11.59 Function or

Synopsis: or(...) returns type **string**

Calculate whether any argument evaluates to true
Example:

```
commands:
  "/usr/bin/generate_config $(config)"
    ifvarclass => or(not(fileexists("/etc/config/$(config)")), "force_configs");
```

Notes:

History: Was introduced in 3.2.0, Nova 2.1.0 (2011)

11.60 Function parseintarray

Synopsis: parseintarray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : A string to parse for input data, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of integers from a file and assign the dimension to a variable

Example:

```
bundle agent test(f)
{
vars:

 #######################################
 # Define data inline for convenience
 #######################################

  "table"   string =>

"1:2
3:4
5:6";

#######################################

 "dim" int => parseintarray(
                              "items",
                               "$(table)",
                               "\s*#[^\n]*",
                               ":",
                               "1000",
                               "200000"
                               );

 "keys" slist => getindices("items");

reports:
  cfengine_3::
         "$(keys)";
}
```

Notes:

History: Was introduced in version 3.1.5a1, Nova 2.1.0 (2011)

This function mirrors the exact behaviour of `readintarray()`, but reads data from a variable instead of a file see Section 11.70 [Function readintarray], page 473. By making data readable from a variable, data driven policies can be kept inline. This means that they will be visible in the CFEngine Knowledge Management portal.

11.61 Function parserealarray

Synopsis: parserealarray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : A string to parse for input data, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of real numbers from a file and assign the dimension to a variable

Example:

```
bundle agent test(f)
{
vars:

 #######################################
 # Define data inline for convenience
 #######################################

  "table"   string =>

"1:1.6
2:2.5
3:3.4";

#######################################

 "dim" int => parserealarray(
                                "items",
                                  "$(table)",
                                  "\s*#[^\n]*",
```

```
                                 ":",
                                 "1000",
                                 "200000"
                                 );

 "keys" slist => getindices("items");

reports:
  cfengine_3::
         "$(keys)";
}
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1.0 (2011)

This function mirrors the exact behaviour of `readrealarray()`, but reads data from a variable instead of a file see Section 11.72 [Function readrealarray], page 475. By making data readable from a variable, data driven policies can be kept inline. This means that they will be visible in the CFEngine Knowledge Management portal.

11.62 Function parsestringarray

Synopsis: parsestringarray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : A string to parse for input data, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of strings from a file and assign the dimension to a variable

Example:

```
bundle agent test(f)
{
vars:

 ########################################
 # Define data inline for convenience
 ########################################

  "table"   string =>
```

```
"one: a
two: b
three: c";

########################################

 "dim" int => parsestringarray(
                                  "items",
                                   "$(table)",
                                   "\s*#[^\n]*",
                                   ":",
                                   "1000",
                                   "200000"
                                   );

 "keys" slist => getindices("items");

reports:
  cfengine_3::
         "$(keys)";
}
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1.0 (2011)

This function mirrors the exact behaviour of `readstringarray()`, but reads data from a variable instead of a file see Section 11.74 [Function readstringarray], page 477. By making data readable from a variable, data driven policies can be kept inline. This means that they will be visible in the CFEngine Knowledge Management portal.

11.63 Function parsestringarrayidx

Synopsis: parsestringarrayidx(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : A string to parse for input data, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of strings from a file and assign the dimension to a variable with integer indeces
Example:

```
bundle agent test(f)
{
vars:

 #######################################
 # Define data inline for convenience
 #######################################

  "table"   string =>

"one: a
two: b
three: c";

#######################################

 "dim" int => parsestringarrayidx(
                               "items",
                                 "$(table)",
                                 "\s*#[^\n]*",
                                 ":",
                                 "1000",
                                 "200000"
                                 );

 "keys" slist => getindices("items");

reports:
  cfengine_3::
         "$(keys)";
}
```

Notes:

History: Was introduced in version 3.1.5, Nova 2.1.0 (2011)

This function mirrors the exact behaviour of `readstringarrayidx()`, but reads data from a variable instead of a file see Section 11.75 [Function readstringarrayidx], page 479. By making data readable from a variable, data driven policies can be kept inline. This means that they will be visible in the CFEngine Knowledge Management portal.

11.64 Function peers

Synopsis: peers(arg1,arg2,arg3) returns type **slist**

arg1 : File name of host list, *in the range* "?(/.*)

arg2 : Comment regex pattern, *in the range* .*
arg3 : Peer group size, *in the range* 0,99999999999

Get a list of peers (not including ourself) from the partition to which we belong

Example:

```
bundle agent peers
{
vars:

  "mygroup" slist => peers("/tmp/hostlist","#.*",4);

  "myleader" string => peerleader("/tmp/hostlist","#.*",4);

  "all_leaders" slist => peerleaders("/tmp/hostlist","#.*",4);

reports:

 linux::

   "mypeer $(mygroup)";
   "myleader $(myleader)";
   "another leader $(all_leaders)";

}
```

Notes:

```
(slist) peers(file of hosts,comment pattern,group size);
```

This function returns a list of hostnames that may be considered peers of the current host. Peers are defined according to a list of hosts, provided as a file in the first argument. This file should contain a list (one per line), possible with comments, of fully qualified host names. CFEngine breaks up this list into non-overlapping groups of up to *groupsize*, each of which has a leader which is the first host in the group.

The current host should belong to this file if it is expected to interact with the others. The function returns nothing if the host does not belong to the list.

ARGUMENTS:

'File of hosts'
: A path to a list of hosts.

'Comment pattern'
: A pattern that matches a legal comment in the file. The regex may match a partial line (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). Comments are stripped as the file is read.

'Group size'
: A number between 2 and 64 which represents the number of peers in a peer-group. An arbitary limit of 64 is set on groups to avoid nonsensical promises.

Example file:

```
one
two
three # this is a comment
four
five
six
seven
eight
nine
ten
eleven
twelve
etc
```

11.65 Function peerleader

Synopsis: peerleader(arg1,arg2,arg3) returns type **string**

arg1 : File name of host list, *in the range* "?(/.*)
arg2 : Comment regex pattern, *in the range* .*
arg3 : Peer group size, *in the range* 0,99999999999

Get the assigned peer-leader of the partition to which we belong

Example:

```
bundle agent peers
{
vars:

  "mygroup" slist => peers("/tmp/hostlist","#.*",4);

  "myleader" string => peerleader("/tmp/hostlist","#.*",4);

  "all_leaders" slist => peerleaders("/tmp/hostlist","#.*",4);

reports:

 linux::
```

```
    "mypeer $(mygroup)";
    "myleader $(myleader)";
    "another leader $(all_leaders)";

}
```

Notes:

```
(string) peerleader(file of hosts,comment pattern,group size);
```

This function returns the name of a ost that may be considered the leader of a group of peers of the current host. Peers are defined according to a list of hosts, provided as a file in the first argument. This file should contain a list (one per line), possibly with comments, of fully qualified host names. CFEngine breaks up this list into non-overlapping groups of up to *groupsize*, each of which has a leader which is the first host in the group.

The current host should belong to this file if it is expected to interact with the others. The function returns nothing if the host does not belong to the list.

ARGUMENTS:

'File of hosts'
: A path to a list of hosts.

'Comment pattern'
: A pattern that matches a legal comment in the file. The regex may match a partial line (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). Comments are stripped as the file is read.

'Group size'
: A number between 2 and 64 which represents the number of peers in a peer-group. An arbitary limit of 64 is set on groups to avoid nonsensical promises.

Example file:

```
one
two
three # this is a comment
four
five
six
seven
eight
nine
ten
eleven
twelve
etc
```

11.66 Function peerleaders

Synopsis: peerleaders(arg1,arg2,arg3) returns type **slist**

arg1 : File name of host list, *in the range* "?(/.*)
arg2 : Comment regex pattern, *in the range* .*
arg3 : Peer group size, *in the range* 0,99999999999

Get a list of peer leaders from the named partitioning

Example:

```
bundle agent peers
{
vars:

  "mygroup" slist => peers("/tmp/hostlist","#.*",4);

  "myleader" string => peerleader("/tmp/hostlist","#.*",4);

  "all_leaders" slist => peerleaders("/tmp/hostlist","#.*",4);

reports:

 linux::

   "mypeer $(mygroup)";
   "myleader $(myleader)";
   "another leader $(all_leaders)";

}
```

Notes:

```
(slist) peers(file of hosts,comment pattern,group size);
```

This function returns a list of hostnames that may be considered peer leaders in the partitioning scheme described in the file of hosts. Peers are defined according to a list of hosts, provided as a file in the first argument. This file should contain a list (one per line), possible with comments, of fully qualified host names. CFEngine breaks up this list into non-overlapping groups of up to *groupsize*, each of which has a leader which is the first host in the group.

The current host need not belong to this file.

ARGUMENTS:

'File of hosts'
: A path to a list of hosts.

'Comment pattern'
A pattern that matches a legal comment in the file. The regex may match a partial line (that is, it is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). Comments are stripped as the file is read.

'Group size'
A number between 2 and 64 which represents the number of peers in a peer-group. An arbitary limit of 64 is set on groups to avoid nonsensical promises.

Example file:

```
one
two
three # this is a comment
four
five
six
seven
eight
nine
ten
eleven
twelve
etc
```

11.67 Function product

Synopsis: product(arg1) returns type **real**

arg1 : A list of arbitrary real values, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Return the product of a list of reals

Example:

```
bundle agent test
{
vars:

  "series" rlist => { "1.1", "2.2", "3.3", "5.5", "7.7" };

  "prod" real => product("series");
  "sum"  real => sum("series");

reports:

  cfengine_3::

    "Product result: $(prod) > $(sum)";
}
```

Notes:

Of course, you could easily combine `product` with `readstringarray` or `readreallist`, etc. to collect summary information from a source external to CFEngine.

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

This function might be used for simple ring computation.

11.68 Function randomint

Synopsis: randomint(arg1,arg2) returns type **int**

arg1 : Lower inclusive bound, *in the range* -99999999999,9999999999
arg2 : Upper inclusive bound, *in the range* -99999999999,9999999999

Generate a random integer between the given limits

Example:

```
vars:

 "ran"    int => randomint(4,88);
```

Notes:

The limits must be integer values and the resulting numbers are based on the entropy of the md5 algorithm.

11.69 Function readfile

Synopsis: readfile(arg1,arg2) returns type **string**

arg1 : File name, *in the range* "?(/.*)
arg2 : Maximum number of bytes to read, *in the range* 0,99999999999

Read max number of bytes from named file and assign to variable

Example:

```
vars:

 "xxx"

    string => readfile( "/home/mark/tmp/testfile" , "33" );
```

Notes:

The file (fragment) is read into a single scalar variable.

11.70 Function readintarray

Synopsis: readintarray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : File name to read, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of integers from a file and assign the dimension to a variable

Example:

```
vars:

  "dim_array"

     int =>  readintarray("array_name","/tmp/array","#[^\n]*",":",10,4000);
```

ARGUMENTS:

'array_name'
: The name to be used for the container array (the array is filled by this routine).

'filename'
: The name of a text file containing the text to be split up as a list.

'comment'
: A regex pattern which specifies comments to be ignored in the file. The `comment` field will strip out unwanted patterns from the file being read, leaving unstripped characters to be split into fields. Using the empty string ("") indicates no commments. The regex is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

'split'
: A regex pattern which is used to parse the field separator(s) to split up the file into items. The `split` regex is also unanchored.

'maxent'
: The maximum number of list items to read from the file

'maxsize'
: The maximum number of bytes to read from the file

Notes:

Reads a two dimensional array from a file. One dimension is separated by the character specified in the argument, the other by the the lines in the file. The first field of the lines names the first array argument.

```
1: 5:7:21:13
2:19:8:14:14
3:45:1:78:22
4:64:2:98:99
```

Results in

```
array_name[1][0]   1
array_name[1][1]   5
array_name[1][2]   7
array_name[1][3]   21
array_name[1][4]   13
array_name[2][0]   2
array_name[2][1]   19
array_name[2][2]   8
array_name[2][3]   14
array_name[2][4]   14
array_name[3][0]   3
array_name[3][1]   45
array_name[3][2]   1
array_name[3][3]   78
array_name[3][4]   22
array_name[4][0]   4
array_name[4][1]   64
array_name[4][2]   2
array_name[4][3]   98
array_name[4][4]   99
```

11.71 Function readintlist

Synopsis: readintlist(arg1,arg2,arg3,arg4,arg5) returns type **ilist**

arg1 : File name to read, *in the range* "?(/.*)
arg2 : Regex matching comments, *in the range* .*
arg3 : Regex to split data, *in the range* .*
arg4 : Maximum number of entries to read, *in the range* 0,99999999999
arg5 : Maximum bytes to read, *in the range* 0,99999999999

Read and assign a list variable from a file of separated ints
Example:

```
body common control

{
bundlesequence  => { "example" };
}
```

```
#############################################################

bundle agent example

{
vars:

  "mylist" ilist => { readintlist("/tmp/listofint","#.*","[\n]",10,400) };

reports:

  Yr2008::

    "List entry: $(mylist)";

}
```

ARGUMENTS:

'filename' The name of a text file containing text to be split up as a list.

'comment' A regex pattern which specifies comments to be ignored in the file. The comment field will strip out unwanted patterns from the file being read, leaving unstripped characters to be split into fields. Using the empty string ("") indicates no commments. The regex is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

'split' A regex pattern which is used to parse the field separator(s) to split up the file into items. The split regex is also unanchored.

'maxent' The maximum number of list items to read from the file

'maxsize' The maximum number of bytes to read from the file

Notes:

11.72 Function readrealarray

Synopsis: readrealarray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : File name to read, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of real numbers from a file and assign the dimension to a variable

Example:

```
vars:

  "dim_array"

     int =>  readrealarray("array_name","/tmp/array","#[^\n]*",":",10,4000);
```

ARGUMENTS:

'array_name'
: The name to be used for the container array (the array is filled by this routine).

'filename'
: The name of a text file containing the text to be split up as a list.

'comment'
: A regex pattern which specifies comments to be ignored in the file. The `comment` field will strip out unwanted patterns from the file being read, leaving unstripped characters to be split into fields. Using the empty string (`""`) indicates no commments. The regex is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

'split'
: A regex pattern which is used to parse the field separator(s) to split up the file into items. The `split` regex is also unanchored.

'maxent'
: The maximum number of list items to read from the file

'maxsize'
: The maximum number of bytes to read from the file

Notes:

For detailed notes, See Section 11.70 [Function readintarray], page 473.

11.73 Function readreallist

Synopsis: readreallist(arg1,arg2,arg3,arg4,arg5) returns type **rlist**

arg1 : File name to read, *in the range* "?(/.*)
arg2 : Regex matching comments, *in the range* .*
arg3 : Regex to split data, *in the range* .*
arg4 : Maximum number of entries to read, *in the range* 0,99999999999
arg5 : Maximum bytes to read, *in the range* 0,99999999999

Read and assign a list variable from a file of separated real numbers
Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "mylist" rlist => { readreallist("/tmp/listofreal","#.*","[\n]",10,400) };

reports:

  Yr2008::

    "List entry: $(mylist)";

}
```

ARGUMENTS:

'filename' The name of a text file containing text to be split up as a list.

'comment' A regex pattern which specifies comments to be ignored in the file. The comment field will strip out unwanted patterns from the file being read, leaving unstripped characters to be split into fields. Using the empty string ("") indicates no commments. The regex is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

'split' A regex pattern which is used to parse the field separator(s) to split up the file into items. The split regex is also unanchored.

'maxent' The maximum number of list items to read from the file

'maxsize' The maximum number of bytes to read from the file

Notes:

11.74 Function readstringarray

Synopsis: readstringarray(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : File name to read, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999

arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of strings from a file and assign the dimension to a variable

Example:

```
vars:

  "dim_array"

     int =>  readstringarray("array_name","/tmp/array","\s*#[^\n]*",":",10,4000);
```

Returns an integer number of keys in the array (i.e., the number of lines matched). If you only want the fields in the first matching line (e.g., to mimic the behavior of the *getpwnam(3)* on the file '`/etc/passwd`'), See Section 11.21 [Function getfields], page 431, instead.

ARGUMENTS:

'`array_name`'
: The name to be used for the container array (the array is filled by this routine).

'`filename`'
: The name of a text file containing the text to be split up as a list.

'`comment`'
: A regex pattern which specifies comments to be ignored in the file. The `comment` field will strip out unwanted patterns from the file being read, leaving unstripped characters to be split into fields. Using the empty string (`""`) indicates no commments. The regex is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

'`split`'
: A regex pattern which is used to parse the field separator(s) to split up the file into items. The `split` regex is also unanchored.

'`maxent`'
: The maximum number of list items to read from the file

'`maxsize`'
: The maximum number of bytes to read from the file

Notes:

Reads a two dimensional array from a file. One dimension is separated by the character specified in the argument, the other by the the lines in the file. The first field of the lines names the first array argument.

```
at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
avahi:x:103:105:User for Avahi:/var/run/avahi-daemon:/bin/false # Disallow login
beagleindex:x:104:106:User for Beagle indexing:/var/cache/beagle:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
# Daemon has the default shell
daemon:x:2:2:Daemon:/sbin:
```

Results in a systematically indexed map of the file. Some samples are show below to illustrate the pattern.

```
...
array_name[daemon][0]   daemon
```

```
array_name[daemon][1]   x
array_name[daemon][2]   2
array_name[daemon][3]   2
array_name[daemon][4]   Daemon
array_name[daemon][5]   /sbin
array_name[daemon][6]   /bin/bash
...
array_name[at][3]       25
array_name[at][4]       Batch jobs daemon
array_name[at][5]       /var/spool/atjobs
array_name[at][6]       /bin/bash
...
array_name[games][3]    100
array_name[games][4]    Games account
array_name[games][5]    /var/games
array_name[games][6]    /bin/bash
...
```

11.75 Function readstringarrayidx

Synopsis: readstringarrayidx(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : Array identifier to populate, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : A string to parse for input data, *in the range* "?(/.*)
arg3 : Regex matching comments, *in the range* .*
arg4 : Regex to split data, *in the range* .*
arg5 : Maximum number of entries to read, *in the range* 0,99999999999
arg6 : Maximum bytes to read, *in the range* 0,99999999999

Read an array of strings from a file and assign the dimension to a variable with integer indeces
Example:

```
vars:

  "dim_array"

     int =>  readstringarrayidx("array_name","/tmp/array","\s*#[^\n]*",":",10,4000);
```

Returns an integer number of keys in the array (i.e., the number of lines matched). If you only want the fields in the first matching line (e.g., to mimic the behavior of the *getpwnam(3)* on the file '/etc/passwd'), See Section 11.21 [Function getfields], page 431, instead.
ARGUMENTS:

'array_name'
: The name to be used for the container array (the array is filled by this routine).

'filename'
: The name of a text file containing the text to be split up as a list.

'comment'
: A regex pattern which specifies comments to be ignored in the file. The comment field will strip out unwanted patterns from the file being read, leaving unstripped characters to

be split into fields. Using the empty string ("") indicates no commments. The regex is unanchored. See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39.

'split' A regex pattern which is used to parse the field separator(s) to split up the file into items. The split regex is also unanchored.

'maxent' The maximum number of list items to read from the file

'maxsize' The maximum number of bytes to read from the file

Notes:

Reads a two dimensional array from a file. One dimension is separated by the character specified in the argument, the other by the the lines in the file. The array arguments are both integer indeces, allowing for non-identifiers at first field (e.g. duplicates or names with spaces), unlike readstringarray.

```
at spaced:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
duplicate:x:103:105:User for Avahi:/var/run/avahi-daemon:/bin/false # Disallow login
beagleindex:x:104:106:User for Beagle indexing:/var/cache/beagle:/bin/bash
duplicate:x:1:1:bin:/bin:/bin/bash
# Daemon has the default shell
daemon:x:2:2:Daemon:/sbin:
```

Results in a systematically indexed map of the file. Some samples are show below to illustrate the pattern.

```
array_name[0][0]       at spaced
array_name[0][1]       x
array_name[0][2]       25
array_name[0][3]       25
array_name[0][4]       Batch jobs daemon
array_name[0][5]       /var/spool/atjobs
array_name[0][6]       /bin/bash
array_name[1][0]       duplicate
array_name[1][1]       x
array_name[1][2]       103
array_name[1][3]       105
array_name[1][4]       User for Avahi
array_name[1][5]       /var/run/avahi-daemon
array_name[1][6]       /bin/false
...
```

11.76 Function readstringlist

Synopsis: readstringlist(arg1,arg2,arg3,arg4,arg5) returns type **slist**

arg1 : File name to read, *in the range* "?(/.*)
arg2 : Regex matching comments, *in the range* .*
arg3 : Regex to split data, *in the range* .*
arg4 : Maximum number of entries to read, *in the range* 0,99999999999
arg5 : Maximum bytes to read, *in the range* 0,99999999999

Read and assign a list variable from a file of separated strings

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "mylist" slist => readstringlist("/tmp/listofstring", "#.*", "\s", 10, 400);

reports:

  Yr2008::

    "List entry: $(mylist)";

}
```

ARGUMENTS:

'filename' The name of a text file containing text to be split up as a list.

'comment' A regex pattern which specifies comments to be ignored in the file. The `comment` field will strip out unwanted patterns from the file being read, leaving unstripped characters to be split into fields. The regex is unanchored, See Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39. **Note** that the text is not treated as a collection of lines, but is read as a single block of `maxsize` characters, and the regex is applied to that as a single string.

'split' A regex pattern which is used to parse the field separator(s) to split up the file into items. The `split` regex is also unanchored.

'maxent' The maximum number of list items to read from the file

'maxsize' The maximum number of bytes to read from the file

Notes:

The following example file would be split into a list of the first ten Greek letters - alpha through kappa.

```
alpha beta
gamma # This is a comment
delta epsilon zeta
       eta
              theta
iota
kappa          lambda
mu
nu
etc
```

11.77 Function readtcp

Synopsis: readtcp(arg1,arg2,arg3,arg4) returns type **string**

arg1 : Host name or IP address of server socket, *in the range* .*
arg2 : Port number, *in the range* 0,99999999999
arg3 : Protocol query string, *in the range* .*
arg4 : Maximum number of bytes to read, *in the range* 0,99999999999

Connect to tcp port, send string and assign result to variable

Example:

```
bundle agent example

{
vars:

  "my80" string => readtcp("research.iu.hio.no","80","GET /index.php HTTP/1.1$(const.r)$(const.n)Hos

classes:

  "server_ok" expression => regcmp("[^\n]*200 OK.*\n.*","$(my80)");

reports:

  server_ok::

    "Server is alive";

  !server_ok::

    "Server is not responding - got $(my80)";
}
```

'hostnameip'
: The host name or IP address of a tcp socket.

'port' : The port number to connect to.

'sendstring'
: A string to send to the TCP port to elicit a response

'maxbytes' : The maximum number of bytes to read in response.

Important note: not all Unix TCP read operations respond to signals for interruption so poorly formed requests can hang. Always test TCP connections fully before deploying.

Notes:

If the send string is empty, no data are sent or received from the socket. Then the function only tests whether the TCP port is alive and returns an empty variable.

Note that on some systems the timeout mechanism does not seem to successfully interrupt the waiting system calls so this might hang if you send a query string that is incorrect. This should not happen, but the cause has yet to be diagnosed.

11.78 Function regarray

Synopsis: regarray(arg1,arg2) returns type **class**

arg1 : Cfengine array identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : Regular expression, *in the range* .*

True if arg1 matches any item in the associative array with id=arg2

Example:

```
body common control

{
bundlesequence  => { "testbundle"  };
}

###########################################

bundle agent testbundle
{
vars:

  "myarray[0]" string => "bla1";
  "myarray[1]" string => "bla2";
```

```
  "myarray[3]" string => "bla";
  "myarray"    string => "345";
  "not"        string => "345";

classes:

  "ok" expression => regarray("myarray","b.*2");

reports:

 ok::

    "Found in list";

 !ok::

    "Not found in list";

}
```

Notes:

Tests whether an associative array contains elements matching a certain regular expression. The result is a class.

ARGUMENTS:

'array_name'
: The name of the array, with no '$()' surrounding it, etc.

'regex'
: A regular expression to match the content. The regular expression must match the complete array element (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

11.79 Function regcmp

Synopsis: regcmp(arg1,arg2) returns type **class**

arg1 : Regular expression, *in the range* .*
arg2 : Match string, *in the range* .*

True if arg1 is a regular expression matching that matches string arg2

Example:

```
bundle agent subtest(user)
```

```
{
classes:

  "invalid" not => regcmp("[a-z]{4}","$(user)");

reports:

 !invalid::

  "User name $(user) is valid at exactly 4 letters";

 invalid::

  "User name $(user) is invalid";
}
```

Notes:

Compares a string to a regular expression.

ARGUMENTS:

'regex' A regular expression to match the content. The regular expression must match the complete content (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'string' Test data for the regular expression.

If there are multiple-lines in the data, it is necessary to code these explicitly, as regular expressions do not normally match the end of line as a regular character (they only match end of string). You can do this using either standard regular expression syntax or using the additional features of PCRE (where (?ms) changes the way that '.', '^' and '$' behave), e.g.

```
body common control
{
bundlesequence => { "example" };
}

bundle agent example
{
vars:

  "x" string => "
NAME: apache2 - Apache 2.2 web server
CATEGORY: application
ARCH: all
VERSION: 2.2.3,REV=2006.09.01
BASEDIR: /
VENDOR: http://httpd.apache.org/ packaged for CSW by Cory Omand
PSTAMP: comand@thor-20060901022929
INSTDATE: Dec 14 2006 16:05
HOTLINE: http://www.blastwave.org/bugtrack/
```

```
      EMAIL: comand@blastwave.org
      STATUS: completely installed
      ";

      classes:

        "pkg_installed" expression => regcmp("(.*\n)*STATUS:\s+completely installed\n(.*\n)*",$(x));

        "base_is_root" expression => regcmp("(?ms).*^BASEDIR:\s+/$.*", $(x));

      reports:

        pkg_installed::

          "installed";

        base_is_root::

          "in root";
      }
```

11.80 Function regextract

Synopsis: regextract(arg1,arg2,arg3) returns type **class**

arg1 : Regular expression, *in the range* .*
arg2 : Match string, *in the range* .*
arg3 : Identifier for back-references, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

True if the regular expression in arg 1 matches the string in arg2 and sets a non-empty array of backreferences named arg3

Example:

```
bundle agent testbundle
{
classes:

    # Extract regex backreferences and put them in an array

    "ok" expression => regextract(
                                 "xx ([^\s]+) ([^\s]+).* xx",
                                 "xx one two three four xx",
                                 "myarray"
                                 );
reports:

 ok::
```

```
   "ok - \"$(myarray[0])\" = xx + \"$(myarray[1])\" + \"$(myarray[2])\" + .. + xx";
}
```

Notes:

Arguments:

regex A regular expression containing one or more parenthesized back references. The regular expression must match the entire string (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

data A string to be matched to the regular expression.

identifier The name of an array which (if there are any back reference matches from the regular expression) will be populated with the values, in the manner

```
$(identifier[0]) = entire string
$(identifier[1]) = back reference 1, etc
```

History: This function was introduced in CFEngine version 3.0.4 (2010)

11.81 Function registryvalue

Synopsis: registryvalue(arg1,arg2) returns type **string**

arg1 : Windows registry key, *in the range* .*
arg2 : Windows registry value-id, *in the range* .*

Returns a value for an MS-Win registry key,value pair

Example:

```
bundle agent reg
{
vars:

  "value" string => registryvalue("HKEY_LOCAL_MACHINE\SOFTWARE\CFEngine AS\CFEngine","value3");

reports:

  windows::

   "Value extracted: $(value)";

}
```

Notes:

This function applies only to Windows-based systems. It reads a data field for the value named in the second argument, which lies within the registry key given by the first argument.

The value is parsed as a string. Currently values of type `REG_SZ` (string), `REG_EXPAND_SZ` (expandable string) and `REG_DWORD` (double word) are supported.

11.82 Function regline

Synopsis: regline(arg1,arg2) returns type **class**

arg1 : Regular expression, *in the range* .*
arg2 : Filename to search, *in the range* .*

True if the regular expression in arg1 matches a line in file arg2
Example:

```
bundle agent testbundle

{
files:

  "/tmp/testfile" edit_line => test;
}

#########################################################

bundle edit_line test
{
classes:

    "ok" expression => regline(".*XYZ.*","$(edit.filename)");

reports:

 ok::

   "File $(edit.filename) has a line with \"XYZ\" in it";

}
```

Notes:

Note that the regular expression must match an entire line of the file in order to give a true result. This function is useful for `edit_line` applications, where one might want to set a class for detecting the presence of a string which does not exactly match one being inserted. e.g.

```
bundle edit_line upgrade_cfexecd
  {
  classes:

    # Check there is not already a crontab line, not identical to
    # the one proposed below...

    "exec_fix" not => regline(".*cf-execd.*","$(edit.filename)");

  insert_lines:

    exec_fix::

     "0,5,10,15,20,25,30,35,40,45,50,55 * * * * /var/cfengine/bin/cf-execd -F";

  reports:

    exec_fix::

     "Added a 5 minute schedule to crontabs";
  }
```

11.83 Function reglist

Synopsis: reglist(arg1,arg2) returns type **class**

arg1 : Cfengine list identifier, *in the range* @[(][a-zA-Z0-9]+[)]
arg2 : Regular expression, *in the range* .*

True if the regular expression in arg2 matches any item in the list whose id is arg1
Example:

```
vars:

 "nameservers" slist => {
                        "128.39.89.10",
                        "128.39.74.16",
                        "192.168.1.103"
                        };
classes:
```

```
  "am_name_server" expression => reglist("@(nameservers)",escape("$(sys.ipv4[eth0])"));
```

Notes:

Matches a list of test strings to a regular expression. In the example above, the IP address in `$(sys.ipv4[eth0])` must be `escaped`, because if not, the dot ('`.`') characters in the IP address would be interpreted as regular expression "match any" characters.

ARGUMENTS:

'`list`' The list of strings to test with the regular expression.

'`regex`' The scalar regular expression string. The regular expression must match the entire string (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

11.84 Function regldap

Synopsis: regldap(arg1,arg2,arg3,arg4,arg5,arg6,arg7) returns type **class**

arg1 : URI, *in the range* .*
arg2 : Distinguished name, *in the range* .*
arg3 : Filter, *in the range* .*
arg4 : Record name, *in the range* .*
arg5 : Search scope policy, *in the range* subtree,onelevel,base
arg6 : Regex to match results, *in the range* .*
arg7 : Security level, *in the range* none,ssl,sasl

True if the regular expression in arg6 matches a value item in an ldap search

Example:

```
classes:

   "found" expression => regldap(
                                "ldap://ldap.example.org",
                                "dc=cfengine,dc=com",
                                "(sn=User)",
                                "uid",
                                "subtree",
                                "jon.*",
                                "none"
                                );
```

Notes:

```
(class) regldap(uri,dn,filter,name,scope,regex,security)
```

This function retrieves a single field from all matching LDAP records identified by the search parameters and compares it to a regular expression. If there is a match, true is returned else false.

ARGUMENTS:

'uri' String value of the ldap server. e.g. "ldap://ldap.cfengine.com.no"

'dn' Distinguished name, an ldap formatted name built from components, e.g. "dc=cfengine,dc=com".

'filter' String filter criterion, in ldap search, e.g. "(sn=User)".

'name' String value, the name of a single record to be retrieved, e.g. uid.

'scope' Menu option, the type of ldap search, from the specified root. May take values:

```
subtree
onelevel
base
```

'regex' A regular expression string to match to the results of an LDAP search. The regular expression must match the entire named field (that is, it is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). If any item matches the regex, the result will be true.

'security' Menu option indicating the encryption and authentication settings for communication with the LDAP server. These features might be subject to machine and server capabilites.

```
none
ssl
sasl
```

11.85 Function remotescalar

Synopsis: remotescalar(arg1,arg2,arg3) returns type **string**

arg1 : Variable identifier, *in the range* [a-zA-Z0-9_$(){}\[\].:]+
arg2 : Hostname or IP address of server, *in the range* .*
arg3 : Use enryption, *in the range* true,false,yes,no,on,off

Read a scalar value from a remote cfengine server

Example:

```
vars:
```

```
 "remote" string => remotescalar("test_scalar","127.0.0.1","yes");
```

Notes:

The remote system's cf-serverd must accept the query for the requested variable from the host that is requesting it. An example of this configuration follows.

This function asks for an identifier; it is up to the server to interpret what this means and to return a value of its choosing. If the identifier matches a persistent scalar variable (such as is used to count distributed processes in CFEngine Enterprise) then this will be returned preferentially. If no such variable is found, then the server will look for a literal string in a server bundle with a handle that matches the requested object.

```
bundle server access
{
access:
  "value of my test_scalar, can expand variables here - $(sys.host)"
    handle => "test_scalar",
    comment => "Grant access to contents of test_scalar VAR",
    resource_type => "literal",
    admit => { "127.0.0.1" };
}
```

CFEngine caches the value of this variable, so that, if the network is unavailable, the last known value will be used. Hence use of this function is fault tolerant. Care should be taken in attempting to access remote variables that are not available, as the repeated connections needed to resolve the absence of a value can lead to undesirable behaviour. As a general rule, users are recommended to refrain from relying on the availability of network resources.

(string) remotescalar(*resource handle*,*host/IP address*,*encrypt*);

This function downloads a string from a remote server, using the promise handle as a variable identifier. Availability: Enterprise editions of CFEngine only.

ARGUMENTS:

'`resource handle`'
: The name of the promise on the server side

'`host or IP address`'
: The location of the server on which the resource resides.

'`encrypt`'
: Whether to encrypt the connection to the server.

```
true
yes
false
no
```

Note that this function assumes that you have already performed a successful key exchange between systems, (e.g. using either a remote copy or `cf-runagent` connection). It contains no mechanism for trust establishment and will fail if there is no trust relationship pre-established.

11.86 Function remoteclassesmatching

Synopsis: remoteclassesmatching(arg1,arg2,arg3,arg4) returns type **class**

arg1 : Regular expression, *in the range* .*
arg2 : Server name or address, *in the range* .*
arg3 : Use encryption, *in the range* true,false,yes,no,on,off
arg4 : Return class prefix, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Read persistent classes matching a regular expression from a remote cfengine server and add them into local context with prefix
Example:

```
"succeeded" expression => remoteclassesmatching("regex","server","yes","myprefix");
```

Notes:

This function is only available in Enterprise versions of CFEngine (Nova, Enterprise, etc).

This function contacts a remote `cf-serverd` and requests access to defined *persistent classes* on that system. These must be granted access to by making an `access` promise with `resource_type` set to `context`.

The return value is true (sets the class) if communication with the server was successful and classes are populated in the current bundle with a prefix of your choosing. The arguments are:

Regular expression
: This should match a list of *persistent* classes of be returned from the server, if the server is willing, i.e. has granted access to them.

Server
: The name or IP address of the remote server.

Encryption
: Boolean value, whether or not to encrypt communication.

Prefix
: A string to be added to the returned classes, e.g. if the server defines a persistent class '`alpha`', then this would generate a private class in the current bundle called '`myprefix_alpha`'.

Note that this function assumes that you have already performed a successful key exchange between systems, (e.g. using either a remote copy or `cf-runagent` connection). It contains no mechanism for trust establishment and will fail if there is no trust relationship pre-established.

11.87 Function returnszero

Synopsis: returnszero(arg1,arg2) returns type **class**

arg1 : Fully qualified command path, *in the range* "?(/.*)
arg2 : Shell encapsulation option, *in the range* useshell,noshell

True if named shell command has exit status zero

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "my_result" expression => returnszero("/usr/local/bin/mycommand","noshell");

reports:

  !my_result::

    "Command failed";

}
```

Notes:

This is the complement of `execresult`, but it returns a class result rather than the output of the command.

11.88 Function rrange

Synopsis: rrange(arg1,arg2) returns type **rrange [real,real]**

arg1 : Real number, *in the range* -9.99999E100,9.99999E100
arg2 : Real number, *in the range* -9.99999E100,9.99999E100

Define a range of real numbers for cfengine internal use

Example:

?

Notes:

This is not yet used.

11.89 Function selectservers

Synopsis: selectservers(arg1,arg2,arg3,arg4,arg5,arg6) returns type **int**

arg1 : The identifier of a cfengine list of hosts or addresses to contact, *in the range* @[(][a-zA-Z0-9]+[)]
arg2 : The port number, *in the range* 0,99999999999
arg3 : A query string, *in the range* .*
arg4 : A regular expression to match success, *in the range* .*
arg5 : Maximum number of bytes to read from server, *in the range* 0,99999999999
arg6 : Name for array of results, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Select tcp servers which respond correctly to a query and return their number, set array of names
Example:

```
body common control

{
bundlesequence  => { "test"  };
}

###########################################################

bundle agent test

{
vars:

 "hosts" slist => { "slogans.iu.hio.no", "eternity.iu.hio.no", "nexus.iu.hio.no" };
 "fhosts" slist => { "www.cfengine.com", "www.cfengine.org" };

 "up_servers" int =>  selectservers("@(hosts)","80","","","100","alive_servers");
 "has_favicon" int =>
             selectservers(
                 "@(hosts)", "80",
                  "GET /favicon.ico HTTP/1.0$(const.n)Host: www.cfengine.com$(const.n)$(const.n)",
                  "(?s).*OK.*",
                  "200", "favicon_servers");
```

```
classes:

  "someone_alive" expression => isgreaterthan("$(up_servers)","0");

  "has_favicon" expression => isgreaterthan("$(has_favicon)","0");

reports:

  cfengine_3::
    "Number of active servers $(up_servers)";

  someone_alive::
    "First server $(alive_servers[0]) fails over to $(alive_servers[1])";

  has_favicon::
    "At least $(favicon_servers[0]) has a favicon.ico";

}
```

Notes:

This function selects all the TCP ports that are active and functioning from an ordered list and builds an array of their names. This allows us to select a current list of failover alternatives that are pretested.

'hostlist' A list of host names or IP addresses to attempt to connect to.

'port' The port number for the service.

'sendstr' An optional string to send to the server to elicit a response. If `sendstr` is empty, then no query is sent to the server.

'regex_on_reply'
: If a string is sent, this regex must match the entire resulting reply (that is, the regex is anchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39). If there is a multi-line response from the server, special care must be taken to ensure that you match the newlines, too (note the use of `(?s)` in the example above, which allows '`.`' to also match newlines in the multi-line HTTP response). If `regex_on_reply` is empty, then no reply-checking is performed (and any server reply is deemed to be satisfactory).

'maxbytesread_reply'
: The maximum number of bytes to read as the server's reply.

'array_name'
: The name of the array to build containing the names of hosts that pass the above tests. The array is ordered `array_name[0],..` etc.

11.90 Function splayclass

Synopsis: splayclass(arg1,arg2) returns type **class**

arg1 : Input string for classification, *in the range* .*
arg2 : Splay time policy, *in the range* daily,hourly

True if the first argument's time-slot has arrived, according to a policy in arg2

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "my_result" expression => splayclass("$(sys.host)$(sys.ipv4)","daily");

reports:

  my_result::

    "Load balanced class activated";

}
```

Notes:

The lvalue class evaluates to true if the system clock lies within a scheduled time-interval that maps to a hash of the first argument (which may be any arbitrary string). Different strings will hash to different time intervals, and thus one can map different tasks to time-intervals.

This function may be used to distribute a task, typically in multiple hosts, in time over a day or an hourly period, depending on the policy in the second argument (which must be one of `"daily"` or `"hourly"`). This is useful for copying resources to multiple hosts from a single server, (e.g. large software updates), when simultaneous scheduling would lead to a bottleneck and/or server overload.

The function is similar to the `splaytime` feature in `cf-execd`, except that it allows you to base the decision on any string-criterion on a given host. The entropy (or string-variation) in the first argument determines how effectively CFEngine will be able to distribute tasks. CFEngine instances with the same first argument will yield a true result at the same time (and different first argument will yield a true result at a different time). Thus tasks could be scheduled according to group names for predictability, or according to IP addresses for distribution across the policy interval.

The times at which the splayclass will be defined depends on the second argument. If the first argument is `"hourly"` then the class will be defined for a 5-minute interval every hour (and if the first argument is `"daily"`, then the class will be defined for one 5-minute interval every day. This means that `splayclass` assumes that you are running CFEngine with the default schedule of "every 5 minutes". If you change the executor `schedule` control variable, you may prevent the splayclass from ever being defined (that is, if the hashed 5-minute interval that is selected by the splayclass is a time when you have told CFEngine *not* to run).

11.91 Function splitstring

Synopsis: splitstring(arg1,arg2,arg3) returns type **slist**

arg1 : A data string, *in the range* .*
arg2 : Regex to split on, *in the range* .*
arg3 : Maximum number of pieces, *in the range* 0,99999999999

Convert a string in arg1 into a list of max arg3 strings by splitting on a regular expression in arg2

Example:

```
bundle agent test

{
vars:

  "split1" slist => splitstring("one:two:three",":","10");
  "split2" slist => splitstring("one:two:three",":","1");
  "split3" slist => splitstring("alpha:xyz:beta","xyz","10");

reports:

 linux::

  "split1: $(split1)";    # will list "one", "two", and "three"
  "split2: $(split2)";    # will list "one" and "two:three"
  "split3: $(split3)";    # will list "alpha:" and ":beta"

}
```

Notes:

Returns a list of strings from a string.

ARGUMENTS:

'string' The string to be split.

'regex' A regex pattern which is used to parse the field separator(s) to split up the file into items. The regex is unanchored, see Section 2.11.4 [Anchored vs. unanchored regular expressions], page 39).

'maxent' The maximum number of splits to perform.

If the maximum number of splits is insufficient to accomodate all entries, the final entry in the slist that is generated will contain the rest of the unsplit string.

11.92 Function strcmp

Synopsis: strcmp(arg1,arg2) returns type **class**

arg1 : String, *in the range* .*
arg2 : String, *in the range* .*

True if the two strings match exactly

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "same" expression => strcmp("test","test");

reports:

  same::

    "Strings are equal";
```

```
 !same::

    "Strings are not equal";
}
```

Notes:

11.93 Function sum

Synopsis: sum(arg1) returns type **real**

arg1 : A list of arbitrary real values, *in the range* [a-zA-Z0-9_$(){}\[\].:]+

Return the sum of a list of reals

Example:

```
body common control
{
bundlesequence => { "test" };
}

##############################

bundle agent test
{
vars:
  "adds_to_six" ilist => { "1", "2", "3" };
  "six" real => sum("adds_to_six");
  "adds_to_zero" rlist => { "1.0", "2", "-3e0" };
  "zero" real => sum("adds_to_zero");

reports:
  cfengine_3::
    "six is $(six), zero is $(zero)";
}
```

Because `$(six)` and `$(zero)` are both real numbers, the report that is generated will be:

```
six is 6.000000, zero is 0.000000
```

Notes:

Of course, you could easily combine `sum` with `readstringarray` or `readreallist`, etc. to collect summary information from a source external to CFEngine.

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010)

This function might be used for simple ring computation.

11.94 Function translatepath

Synopsis: translatepath(arg1) returns type **string**

arg1 : Unix style path, *in the range* "?(/.*)

Translate path separators from Unix style to the host's native

Example:

```
body common control
{
bundlesequence => { "test" };
}

##############################

bundle agent test
{
vars:
  "inputs_dir" string => translatepath("$(sys.workdir)/inputs");

reports:

  windows::
    "The path has backslashes: $(inputs_dir)";

  !windows::
    "The path has slashes: $(inputs_dir)";
}
```

Notes:

Takes a string argument with slashes as path separators and translate these to the native format for path separators on the host. For example translatepath("a/b/c") would yield "a/b/c" on Unix platforms, but "a\b\c" on Windows.

Be careful when using this function in combination with regular expressions, since backslash is also used as escape character in regex's. For example, in the regex '`dir/.abc`', the dot represents the regular expression "any character", while in the regex '`dir\.abc`', the backslash-dot represents a literal dot character.

11.95 Function usemodule

Synopsis: usemodule(arg1,arg2) returns type **class**

arg1 : Name of module command, *in the range* .*
arg2 : Argument string for the module, *in the range* .*

Execute cfengine module script and set class if successful

Example:

```
body common control
   {
   any::

      bundlesequence  => {
                         test
                         };
   }

###################################################################

bundle agent test

{
classes:

  # returns $(user)

  "done" expression => usemodule("getusers","");

commands:

  "/bin/echo" args => "test $(user)";
}
```

Notes:

Modules must reside in '`WORKDIR/modules`' but no longer require a special naming convention.

ARGUMENTS:

'`Module name`'
: The name of the module without its leading path, since it is assuemed to be in the registered modules directory.

'Argument string'
Any command link arguments to pass to the module.

11.96 Function userexists

Synopsis: userexists(arg1) returns type **class**

arg1 : User name or identifier, *in the range* .*

True if user name or numerical id exists on this host
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "ok" expression => userexists("root");

reports:

  ok::

    "Root exists";

 !ok::

    "Root does not exist";
}
```

Notes:

Checks whether the user is in the password database for the current host. The argument must be a user name or user id.

12 Special Variables

12.1 Variable context const

CFEngine defines a number of variables for embedding unprintable values or values with special meanings in strings.

12.1.1 Variable const.dollar

```
reports:

  some::

   # This will report: The value of $(const.dollar) is $
   "The value of $(const.dollar)(const.dollar) is $(const.dollar)";

   # This will report: But the value of \$(dollar) is \$(dollar)
   "But the value of \$(dollar) is \$(dollar)";
```

12.1.2 Variable const.endl

```
reports:

 cfengine_3::

  "A newline with either $(const.n) or with $(const.endl) is ok";
  "But a string with \n in it does not have a newline!";
```

12.1.3 Variable const.n

```
reports:

 cfengine_3::

  "A newline with either $(const.n) or with $(const.endl) is ok";
  "But a string with \n in it does not have a newline!";
```

12.1.4 Variable const.r

```
reports:

 cfengine_3::

  "A carriage return character is $(const.r)";
```

12.1.5 Variable const.t

```
reports:

 cfengine_3::

  "A report with a$(const.t)tab in it";
```

12.2 Variable context edit

This context 'edit' is used to access information about editing promises during their execution. It is context dependent and not universally meaningful or available. For example:

```
bundle agent testbundle
{
files:

  "/tmp/testfile"
     edit_line => test;
}

#

bundle edit_line test
{
classes:
    "ok" expression => regline(".*mark.*","$(edit.filename)");

reports:
```

```
  ok::
   "File matched $(edit.filename)";
}
```

$(edit.filename)

This variable points to the filename of the file currently making an edit promise. If the file has been arrived at through a search, this could be different from the 'files' promiser.

12.2.1 Variable edit.filename

This variable points to the filename of the file currently making an edit promise. If the file has been arrived at through a search, this could be different from the 'files' promiser.

12.3 Variable context match

Each time CFEngine matches a string, these values are assigned to a special variable context $(match.*n*). The fragments can be referred to in the remainder of the promise. There are two places where this makes sense. One is in pattern replacement during file editing, and the other is in searching for files.

Consider the examples below:

```
bundle agent testbundle

{
files:

  "/home/mark/tmp/(cf[23])_(.*)"
       create    => "true",
       edit_line => myedit("second $(match.2)");


  # but more specifically...

  "/home/mark/tmp/cf3_(test)"
       create    => "true",
       edit_line => myedit("second $(match.1)");


}
```

12.3.1 Variable match.0

A string matching the complete regular expression whether or not back-references were used in the pattern.

12.4 Variable context `mon`

The variables discovered by `cf-monitord` are placed in this monitoring context. Monitoring variables are expected to be ephemeral properties, rapidly changing.

In enterprise versions of CFEngine, custom defined monitoring targets also become variables in this context, named by the handle of the promise that defined them.

12.4.1 Variable mon.listening_udp4_ports

List variable containing an observational measure collected every 2.5 minutes from cf-monitord, description: port numbers that were observed to be set up to receive connections on the host concerned

12.4.2 Variable mon.listening_tcp4_ports

List variable containing an observational measure collected every 2.5 minutes from cf-monitord, description: port numbers that were observed to be set up to receive connections on the host concerned

12.4.3 Variable mon.listening_udp6_ports

List variable containing an observational measure collected every 2.5 minutes from cf-monitord, description: port numbers that were observed to be set up to receive connections on the host concerned

12.4.4 Variable mon.listening_tcp6_ports

List variable containing an observational measure collected every 2.5 minutes from cf-monitord, description: port numbers that were observed to be set up to receive connections on the host concerned

12.4.5 Variable mon.value_users

Observational measure collected every 2.5 minutes from cf-monitord, description: *Users with active processes - including system users*.

12.4.6 Variable mon.av_users

Observational measure collected every 2.5 minutes from cf-monitord, description: *Users with active processes - including system users*.

12.4.7 Variable mon.dev_users

Observational measure collected every 2.5 minutes from cf-monitord, description: *Users with active processes - including system users*.

12.4.8 Variable mon.value_rootprocs

Observational measure collected every 2.5 minutes from cf-monitord, description: *Sum privileged system processes*.

12.4.9 Variable mon.av_rootprocs

Observational measure collected every 2.5 minutes from cf-monitord, description: *Sum privileged system processes*.

12.4.10 Variable mon.dev_rootprocs

Observational measure collected every 2.5 minutes from cf-monitord, description: *Sum privileged system processes*.

12.4.11 Variable mon.value_otherprocs

Observational measure collected every 2.5 minutes from cf-monitord, description: *Sum non-privileged process*.

12.4.12 Variable mon.av_otherprocs

Observational measure collected every 2.5 minutes from cf-monitord, description: *Sum non-privileged process*.

12.4.13 Variable mon.dev_otherprocs

Observational measure collected every 2.5 minutes from cf-monitord, description: *Sum non-privileged process*.

12.4.14 Variable mon.value_diskfree

Observational measure collected every 2.5 minutes from cf-monitord, description: *Free disk on / partition*.

12.4.15 Variable mon.av_diskfree

Observational measure collected every 2.5 minutes from cf-monitord, description: *Free disk on / partition*.

12.4.16 Variable mon.dev_diskfree

Observational measure collected every 2.5 minutes from cf-monitord, description: *Free disk on / partition*.

12.4.17 Variable mon.value_loadavg

Observational measure collected every 2.5 minutes from cf-monitord, description: *Kernel load average utilization (sum over cores)*.

12.4.18 Variable mon.av_loadavg

Observational measure collected every 2.5 minutes from cf-monitord, description: *Kernel load average utilization (sum over cores)*.

12.4.19 Variable mon.dev_loadavg

Observational measure collected every 2.5 minutes from cf-monitord, description: *Kernel load average utilization (sum over cores)*.

12.4.20 Variable mon.value_netbiosns_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name lookups (in)*.

12.4.21 Variable mon.av_netbiosns_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name lookups (in)*.

12.4.22 Variable mon.dev_netbiosns_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name lookups (in)*.

12.4.23 Variable mon.value_netbiosns_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name lookups (out)*.

12.4.24 Variable mon.av_netbiosns_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name lookups (out)*.

12.4.25 Variable mon.dev_netbiosns_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name lookups (out)*.

12.4.26 Variable mon.value_netbiosdgm_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name datagrams (in)*.

12.4.27 Variable mon.av_netbiosdgm_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name datagrams (in)*.

12.4.28 Variable mon.dev_netbiosdgm_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name datagrams (in)*.

12.4.29 Variable mon.value_netbiosdgm_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name datagrams (out)*.

12.4.30 Variable mon.av_netbiosdgm_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name datagrams (out)*.

12.4.31 Variable mon.dev_netbiosdgm_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *netbios name datagrams (out)*.

12.4.32 Variable mon.value_netbiosssn_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/netbios name sessions (in)*.

12.4.33 Variable mon.av_netbiosssn_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/netbios name sessions (in)*.

12.4.34 Variable mon.dev_netbiosssn_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/netbios name sessions (in)*.

12.4.35 Variable mon.value_netbiosssn_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/netbios name sessions (out)*.

12.4.36 Variable mon.av_netbiosssn_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/netbios name sessions (out)*.

12.4.37 Variable mon.dev_netbiosssn_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/netbios name sessions (out)*.

12.4.38 Variable mon.value_imap_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *imap mail client sessions (in)*.

12.4.39 Variable mon.av_imap_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *imap mail client sessions (in)*.

12.4.40 Variable mon.dev_imap_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *imap mail client sessions (in)*.

12.4.41 Variable mon.value_imap_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *imap mail client sessions (out)*.

12.4.42 Variable mon.av_imap_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *imap mail client sessions (out)*.

12.4.43 Variable mon.dev_imap_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *imap mail client sessions (out)*.

12.4.44 Variable mon.value_cfengine_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *cfengine connections (in)*.

12.4.45 Variable mon.av_cfengine_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *cfengine connections (in)*.

12.4.46 Variable mon.dev_cfengine_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *cfengine connections (in)*.

12.4.47 Variable mon.value_cfengine_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *cfengine connections (out)*.

12.4.48 Variable mon.av_cfengine_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *cfengine connections (out)*.

12.4.49 Variable mon.dev_cfengine_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *cfengine connections (out)*.

12.4.50 Variable mon.value_nfsd_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *nfs connections (in)*.

12.4.51 Variable mon.av_nfsd_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *nfs connections (in)*.

12.4.52 Variable mon.dev_nfsd_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *nfs connections (in)*.

12.4.53 Variable mon.value_nfsd_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *nfs connections (out)*.

12.4.54 Variable mon.av_nfsd_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *nfs connections (out)*.

12.4.55 Variable mon.dev_nfsd_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *nfs connections (out)*.

12.4.56 Variable mon.value_smtp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *smtp connections (in)*.

12.4.57 Variable mon.av_smtp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *smtp connections (in)*.

12.4.58 Variable mon.dev_smtp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *smtp connections (in)*.

12.4.59 Variable mon.value_smtp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *smtp connections (out)*.

12.4.60 Variable mon.av_smtp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *smtp connections (out)*.

12.4.61 Variable mon.dev_smtp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *smtp connections (out)*.

12.4.62 Variable mon.value_www_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *www connections (in)*.

12.4.63 Variable mon.av_www_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *www connections (in)*.

12.4.64 Variable mon.dev_www_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *www connections (in)*.

12.4.65 Variable mon.value_www_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *www connections (out)*.

12.4.66 Variable mon.av_www_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *www connections (out)*.

12.4.67 Variable mon.dev_www_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *www connections (out)*.

12.4.68 Variable mon.value_ftp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ftp connections (in)*.

12.4.69 Variable mon.av_ftp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ftp connections (in)*.

12.4.70 Variable mon.dev_ftp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ftp connections (in)*.

12.4.71 Variable mon.value_ftp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ftp connections (out)*.

12.4.72 Variable mon.av_ftp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ftp connections (out)*.

12.4.73 Variable mon.dev_ftp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ftp connections (out)*.

12.4.74 Variable mon.value_ssh_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ssh connections (in)*.

12.4.75 Variable mon.av_ssh_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ssh connections (in)*.

12.4.76 Variable mon.dev_ssh_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ssh connections (in)*.

12.4.77 Variable mon.value_ssh_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ssh connections (out)*.

12.4.78 Variable mon.av_ssh_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ssh connections (out)*.

12.4.79 Variable mon.dev_ssh_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ssh connections (out)*.

12.4.80 Variable mon.value_wwws_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *wwws connections (in)*.

12.4.81 Variable mon.av_wwws_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *wwws connections (in)*.

12.4.82 Variable mon.dev_wwws_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *wwws connections (in)*.

12.4.83 Variable mon.value_wwws_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *wwws connections (out)*.

12.4.84 Variable mon.av_wwws_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *wwws connections (out)*.

12.4.85 Variable mon.dev_wwws_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *wwws connections (out)*.

12.4.86 Variable mon.value_icmp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ICMP packets (in)*.

12.4.87 Variable mon.av_icmp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ICMP packets (in)*.

12.4.88 Variable mon.dev_icmp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *ICMP packets (in)*.

12.4.89 Variable mon.value_icmp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ICMP packets (out)*.

12.4.90 Variable mon.av_icmp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ICMP packets (out)*.

12.4.91 Variable mon.dev_icmp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *ICMP packets (out)*.

12.4.92 Variable mon.value_udp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *UDP dgrams (in)*.

12.4.93 Variable mon.av_udp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *UDP dgrams (in)*.

12.4.94 Variable mon.dev_udp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *UDP dgrams (in)*.

12.4.95 Variable mon.value_udp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *UDP dgrams (out)*.

12.4.96 Variable mon.av_udp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *UDP dgrams (out)*.

12.4.97 Variable mon.dev_udp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *UDP dgrams (out)*.

12.4.98 Variable mon.value_dns_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *DNS requests (in)*.

12.4.99 Variable mon.av_dns_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *DNS requests (in)*.

12.4.100 Variable mon.dev_dns_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *DNS requests (in)*.

12.4.101 Variable mon.value_dns_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *DNS requests (out)*.

12.4.102 Variable mon.av_dns_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *DNS requests (out)*.

12.4.103 Variable mon.dev_dns_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *DNS requests (out)*.

12.4.104 Variable mon.value_tcpsyn_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP sessions (in)*.

12.4.105 Variable mon.av_tcpsyn_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP sessions (in)*.

12.4.106 Variable mon.dev_tcpsyn_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP sessions (in)*.

12.4.107 Variable mon.value_tcpsyn_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP sessions (out)*.

12.4.108 Variable mon.av_tcpsyn_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP sessions (out)*.

12.4.109 Variable mon.dev_tcpsyn_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP sessions (out)*.

12.4.110 Variable mon.value_tcpack_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP acks (in)*.

12.4.111 Variable mon.av_tcpack_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP acks (in)*.

12.4.112 Variable mon.dev_tcpack_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP acks (in)*.

12.4.113 Variable mon.value_tcpack_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP acks (out)*.

12.4.114 Variable mon.av_tcpack_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP acks (out)*.

12.4.115 Variable mon.dev_tcpack_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP acks (out)*.

12.4.116 Variable mon.value_tcpfin_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP finish (in)*.

12.4.117 Variable mon.av_tcpfin_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP finish (in)*.

12.4.118 Variable mon.dev_tcpfin_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP finish (in)*.

12.4.119 Variable mon.value_tcpfin_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP finish (out)*.

12.4.120 Variable mon.av_tcpfin_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP finish (out)*.

12.4.121 Variable mon.dev_tcpfin_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP finish (out)*.

12.4.122 Variable mon.value_tcpmisc_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP misc (in)*.

12.4.123 Variable mon.av_tcpmisc_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP misc (in)*.

12.4.124 Variable mon.dev_tcpmisc_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP misc (in)*.

12.4.125 Variable mon.value_tcpmisc_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP misc (out)*.

12.4.126 Variable mon.av_tcpmisc_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP misc (out)*.

12.4.127 Variable mon.dev_tcpmisc_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *TCP misc (out)*.

12.4.128 Variable mon.value_webaccess

Observational measure collected every 2.5 minutes from cf-monitord, description: *Webserver hits*.

12.4.129 Variable mon.av_webaccess

Observational measure collected every 2.5 minutes from cf-monitord, description: *Webserver hits*.

12.4.130 Variable mon.dev_webaccess

Observational measure collected every 2.5 minutes from cf-monitord, description: *Webserver hits*.

12.4.131 Variable mon.value_weberrors

Observational measure collected every 2.5 minutes from cf-monitord, description: *Webserver errors*.

12.4.132 Variable mon.av_weberrors

Observational measure collected every 2.5 minutes from cf-monitord, description: *Webserver errors*.

12.4.133 Variable mon.dev_weberrors

Observational measure collected every 2.5 minutes from cf-monitord, description: *Webserver errors*.

12.4.134 Variable mon.value_syslog

Observational measure collected every 2.5 minutes from cf-monitord, description: *New log entries (Syslog)*.

12.4.135 Variable mon.av_syslog

Observational measure collected every 2.5 minutes from cf-monitord, description: *New log entries (Syslog)*.

12.4.136 Variable mon.dev_syslog

Observational measure collected every 2.5 minutes from cf-monitord, description: *New log entries (Syslog)*.

12.4.137 Variable mon.value_messages

Observational measure collected every 2.5 minutes from cf-monitord, description: *New log entries (messages)*.

12.4.138 Variable mon.av_messages

Observational measure collected every 2.5 minutes from cf-monitord, description: *New log entries (messages)*.

12.4.139 Variable mon.dev_messages

Observational measure collected every 2.5 minutes from cf-monitord, description: *New log entries (messages)*.

12.4.140 Variable mon.value_temp0

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 0*.

12.4.141 Variable mon.av_temp0

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 0*.

12.4.142 Variable mon.dev_temp0

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 0*.

12.4.143 Variable mon.value_temp1

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 1*.

12.4.144 Variable mon.av_temp1

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 1*.

12.4.145 Variable mon.dev_temp1

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 1*.

12.4.146 Variable mon.value_temp2

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 2*.

12.4.147 Variable mon.av_temp2

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 2*.

12.4.148 Variable mon.dev_temp2

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 2*.

12.4.149 Variable mon.value_temp3

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 3*.

12.4.150 Variable mon.av_temp3

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 3*.

12.4.151 Variable mon.dev_temp3

Observational measure collected every 2.5 minutes from cf-monitord, description: *CPU Temperature 3*.

12.4.152 Variable mon.value_cpu

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization (all)*.

12.4.153 Variable mon.av_cpu

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization (all)*.

12.4.154 Variable mon.dev_cpu

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization (all)*.

12.4.155 Variable mon.value_cpu0

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 0*.

12.4.156 Variable mon.av_cpu0

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 0*.

12.4.157 Variable mon.dev_cpu0

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 0*.

12.4.158 Variable mon.value_cpu1

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 1*.

12.4.159 Variable mon.av_cpu1

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 1*.

12.4.160 Variable mon.dev_cpu1

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 1*.

12.4.161 Variable mon.value_cpu2

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 2*.

12.4.162 Variable mon.av_cpu2

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 2*.

12.4.163 Variable mon.dev_cpu2

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 2*.

12.4.164 Variable mon.value_cpu3

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 3*.

12.4.165 Variable mon.av_cpu3

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 3*.

12.4.166 Variable mon.dev_cpu3

Observational measure collected every 2.5 minutes from cf-monitord, description: *%CPU utilization 3*.

12.4.167 Variable mon.value_microsoft_ds_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/MS_ds name sessions (in)*.

12.4.168 Variable mon.av_microsoft_ds_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/MS_ds name sessions (in)*.

12.4.169 Variable mon.dev_microsoft_ds_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/MS_ds name sessions (in)*.

12.4.170 Variable mon.value_microsoft_ds_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/MS_ds name sessions (out)*.

12.4.171 Variable mon.av_microsoft_ds_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/MS_ds name sessions (out)*.

12.4.172 Variable mon.dev_microsoft_ds_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Samba/MS_ds name sessions (out)*.

12.4.173 Variable mon.value_www_alt_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Alternative web service connections (in)*.

12.4.174 Variable mon.av_www_alt_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Alternative web service connections (in)*.

12.4.175 Variable mon.dev_www_alt_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Alternative web service connections (in)*.

12.4.176 Variable mon.value_www_alt_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Alternative web client connections (out)*.

12.4.177 Variable mon.av_www_alt_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Alternative web client connections (out)*.

12.4.178 Variable mon.dev_www_alt_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Alternative web client connections (out)*.

12.4.179 Variable mon.value_imaps_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *encrypted imap mail service sessions (in)*.

12.4.180 Variable mon.av_imaps_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *encrypted imap mail service sessions (in)*.

12.4.181 Variable mon.dev_imaps_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *encrypted imap mail service sessions (in)*.

12.4.182 Variable mon.value_imaps_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *encrypted imap mail client sessions (out)*.

12.4.183 Variable mon.av_imaps_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *encrypted imap mail client sessions (out)*.

12.4.184 Variable mon.dev_imaps_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *encrypted imap mail client sessions (out)*.

12.4.185 Variable mon.value_ldap_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service service sessions (in)*.

12.4.186 Variable mon.av_ldap_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service service sessions (in)*.

12.4.187 Variable mon.dev_ldap_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service service sessions (in)*.

12.4.188 Variable mon.value_ldap_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service client sessions (out)*.

12.4.189 Variable mon.av_ldap_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service client sessions (out)*.

12.4.190 Variable mon.dev_ldap_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service client sessions (out)*.

12.4.191 Variable mon.value_ldaps_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service service sessions (in)*.

12.4.192 Variable mon.av_ldaps_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service service sessions (in)*.

12.4.193 Variable mon.dev_ldaps_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service service sessions (in)*.

12.4.194 Variable mon.value_ldaps_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service client sessions (out)*.

12.4.195 Variable mon.av_ldaps_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service client sessions (out)*.

12.4.196 Variable mon.dev_ldaps_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *LDAP directory service client sessions (out)*.

12.4.197 Variable mon.value_mongo_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Mongo database service sessions (in)*.

12.4.198 Variable mon.av_mongo_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Mongo database service sessions (in)*.

12.4.199 Variable mon.dev_mongo_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Mongo database service sessions (in)*.

12.4.200 Variable mon.value_mongo_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Mongo database client sessions (out)*.

12.4.201 Variable mon.av_mongo_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Mongo database client sessions (out)*.

12.4.202 Variable mon.dev_mongo_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Mongo database client sessions (out)*.

12.4.203 Variable mon.value_mysql_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *MySQL database service sessions (in)*.

12.4.204 Variable mon.av_mysql_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *MySQL database service sessions (in)*.

12.4.205 Variable mon.dev_mysql_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *MySQL database service sessions (in)*.

12.4.206 Variable mon.value_mysql_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *MySQL database client sessions (out)*.

12.4.207 Variable mon.av_mysql_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *MySQL database client sessions (out)*.

12.4.208 Variable mon.dev_mysql_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *MySQL database client sessions (out)*.

12.4.209 Variable mon.value_postgres_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *PostgreSQL database service sessions (in)*.

12.4.210 Variable mon.av_postgres_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *PostgreSQL database service sessions (in)*.

12.4.211 Variable mon.dev_postgres_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *PostgreSQL database service sessions (in)*.

12.4.212 Variable mon.value_postgres_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *PostgreSQL database client sessions (out)*.

12.4.213 Variable mon.av_postgres_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *PostgreSQL database client sessions (out)*.

12.4.214 Variable mon.dev_postgres_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *PostgreSQL database client sessions (out)*.

12.4.215 Variable mon.value_ipp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Internet Printer Protocol (in)*.

12.4.216 Variable mon.av_ipp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Internet Printer Protocol (in)*.

12.4.217 Variable mon.dev_ipp_in

Observational measure collected every 2.5 minutes from cf-monitord, description: *Internet Printer Protocol (in)*.

12.4.218 Variable mon.value_ipp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Internet Printer Protocol (out)*.

12.4.219 Variable mon.av_ipp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Internet Printer Protocol (out)*.

12.4.220 Variable mon.dev_ipp_out

Observational measure collected every 2.5 minutes from cf-monitord, description: *Internet Printer Protocol (out)*.

12.5 Variable context sys

System variables are derived from CFEngine's automated discovery of system values. They are provided as variables in order to make automatically adaptive rules for configuration, e.g.

```
files:

 any::

  "$(sys.resolv)"

      create        => "true",
      edit_line     => doresolv("@(this.list1)","@(this.list2)"),
      edit_defaults => reconstruct;
```

The above rule requires no class specification because the variable itself is class-specific.

12.5.1 Variable sys.arch

The variable gives the kernel's short architecture description.

```
# arch = x86_64
```

12.5.2 Variable sys.cdate

The date of the system in canonical form, i.e. in the form of a class.

```
# cdate = Sun_Dec__7_10_39_53_2008_
```

12.5.3 Variable sys.cf_version

The variable gives the version of the running CFEngine Community Edition.

```
# cf_version = 3.0.5
```

12.5.4 Variable sys.class

This variable contains the name of the hard-class category for this host, i.e. its top level operating system type classification.

```
# class = linux
```

12.5.5 Variable sys.cpus

History: Was introduced in 3.3.0, Nova 2.2.0 (2012)

A variable containing the number of CPU cores detected on local host, where possible. This is currently only guaranteed on Linux 2.6 kernels.

```
reports:

 cfengine_3::
```

```
"Number of CPUS = $(sys.cpus)";
```

12.5.6 Variable sys.crontab

The variable gives the location of the current users's master crontab directory.

```
# crontab = /var/spool/crontas/root
```

12.5.7 Variable sys.date

The date of the system as a text string.

```
# date = Sun Dec  7 10:39:53 2008
```

12.5.8 Variable sys.doc_root

History: Was introduced in 3.1.0, Nova 2.0.

A scalar variable containing the default path for the document root of the standard web server package.

12.5.9 Variable sys.domain

The domain name as divined by CFEngine. If the DNS is in use, it could be possible to derive the domain name from its DNS regisration, but in general there is no way to discover this value automatically. The `common control` body permits the ultimate specification of this value.

```
# domain = example.org
```

12.5.10 Variable sys.enterprise_version

History: Was introduced in 3.5.0, Enterprise 3.0.0

The variable gives the version of the running CFEngine Enterprise Edition.

```
# enterprise_version = 3.0.0
```

12.5.11 Variable sys.expires

```
reports:

 nova::

  "License expires $(sys.expires)";
```

12.5.12 Variable sys.exports

The location of the system NFS exports file.

```
# exports = /etc/exports
# exports = /etc/dfs/dfstab
```

12.5.13 Variable sys.flavor

History: Was introduced in 3.2.0, Nova 2.0

A variable containing an operating system identification string that is used to determine the current release of the operating system in a form that can be used as a label in naming. This is used, for instance, to detect which package name to choose when updating software binaries for CFEngine.

This is a synonym for `$(sys.flavour)`.

12.5.14 Variable sys.flavour

History: Was introduced in 3.2.0, Nova 2.0

A variable containing an operating system identification string that is used to determine the current release of the operating system in a form that can be used as a label in naming. This is used, for instance, to detect which package name to choose when updating software binaries for CFEngine.

This is a synonym for `$(sys.flavor)`.

12.5.15 Variable sys.fqhost

The fully qualified name of the host. In order to compute this value properly, the domain name must be defined.

```
# fqhost = host.example.org
```

12.5.16 Variable sys.fstab

The location of the system filesystem (mount) table.

```
# fstab = /etc/fstab
```

12.5.17 Variable sys.hardware_addresses

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

This is a list variable containing a list of all known MAC addresses for system interfaces.

12.5.18 Variable sys.hardware_mac[interface_name]

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

This contains the MAC address of the named interface. e.g.

```
reports:

  linux::

    "Tell me $(harware_mac[eth0])";
```

12.5.19 Variable sys.host

The name of the current host, according to the kernel. It is undefined whether this is qualified or unqualified with a domain name.

```
# host = myhost
```

12.5.20 Variable sys.interface

The assumed (default) name of the main system interface on this host.

```
# interface = eth0
```

12.5.21 Variable sys.interfaces

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

Displays a system list of configured interfaces currently active in use by the system. This list is detected at runtime and it passed in the variables report to a Mission Portal in commercial editions of CFEngine.

To use this list in a policy, you will need a local copy since only local variables can be iterated.

```
bundle agent test
{
vars:

 # To iterate, we need a local copy

 "i1" slist => { @(sys.ip_addresses)} ;
 "i2" slist => { @(sys.interfaces)} ;

reports:

  cfengine::

    "Addresses: $(i1)";
    "Interfaces: $(i2)";
    "Addresses of the interfaces: $(sys.ipv4[$(i2)])";

}
```

12.5.22 Variable sys.ip_addresses

History: Was introduced in 3.3.0, Nova 2.2.0 (2011)

Displays a system list of IP addresses currently in use by the system. This list is detected at runtime and it passed in the variables report to a Mission Portal in commercial editions of CFEngine.

To use this list in a policy, you will need a local copy since only local variables can be iterated.

```
bundle agent test
{
vars:

 # To iterate, we need a local copy

 "i1" slist => { @(sys.ip_addresses)} ;
 "i2" slist => { @(sys.interfaces)} ;
```

```
reports:

  cfengine::

    "Addresses: $(i1)";
    "Interfaces: $(i2)";
    "Addresses of the interfaces: $(sys.ipv4[$(i2)])";

}
```

12.5.23 Variable sys.ipv4

All four octets of the IPv4 address of the first system interface.

Note:

If your system has a single ethernet interface, '$(sys.ipv4)' will contain your IPv4 address. However, if your system has multiple interfaces, then '$(sys.ipv4)' will simply be the IPv4 address of the first interface in the list that has an assigned address, See Section 12.5.24 [Variable sys.ipv4[interface_name]], page 533, for details on obtaining the IPv4 addresses of all interfaces on a system.

12.5.24 Variable sys.ipv4[interface_name]

The full IPv4 address of the system interface named as the associative array index, e.g. '$(ipv4[le0])' or '$(ipv4[xr1])'.

```
# If the IPv4 address on the interfaces are
#        le0 = 192.168.1.101
#         xr1 = 10.12.7.254
#
# Then the octets of all interfaces are accessable as an associative array
# ipv4_1[le0] = 192
# ipv4_2[le0] = 192.168
# ipv4_3[le0] = 192.168.1
#   ipv4[le0] = 192.168.1.101
# ipv4_1[xr1] = 10
# ipv4_2[xr1] = 10.12
# ipv4_3[xr1] = 10.12.7
#   ipv4[xr1] = 10.12.7.254
```

Note:

The list of interfaces may be acquired with 'getindices("sys.ipv4")' (or from any of the other associative arrays). Only those interfaces which are marked as "up" and have an IP address will be listed.

12.5.25 Variable sys.ipv4_1[interface_name]

The first octet of the IPv4 address of the system interface named as the associative array index, e.g. '$(ipv4_1[le0])' or '$(ipv4_1[xr1])', See Section 12.5.24 [Variable sys.ipv4[interface_name]], page 533.

12.5.26 Variable sys.ipv4_2[interface_name]

The first two octets of the IPv4 address of the system interface named as the associative array index, e.g. '$(ipv4_2[le0])' or '$(ipv4_2[xr1])', See Section 12.5.24 [Variable sys.ipv4[interface_name]], page 533.

12.5.27 Variable sys.ipv4_3[interface_name]

The first three octets of the IPv4 address of the system interface named as the associative array index, e.g. '$(ipv4_3[le0])' or '$(ipv4_3[xr1])', See Section 12.5.24 [Variable sys.ipv4[interface_name]], page 533.

12.5.28 Variable sys.license_owner

History: Was introduced in version 3.1.4,Nova 2.0.2 (2011)

```
reports:

 nova::

  "This version of CFEngine is licensed to $(sys.license_owner)";
```

12.5.29 Variable sys.licenses_granted

History: Was introduced in version 3.1.4,Nova 2.0.2 (2011)

```
reports:

 nova::
```

```
  "There are $(sys.licenses_granted) licenses granted for use";
```

12.5.30 Variable sys.licenses_installtime

History: Was introduced in version 3.1.5, Nova 2.1 (2011)

```
reports:

 nova::

  "The license was installed on $(sys.licenses_installtime)";
```

12.5.31 Variable sys.long_arch

The long architecture name for this system kernel. This name is sometimes quite unwieldy but can be useful for logging purposes.

```
# long_arch = linux_x86_64_2_6_22_19_0_1_default__1_SMP_2008_10_14_22_17_43__0200
```

12.5.32 Variable sys.maildir

The name of the system email spool directory.

```
# maildir = /var/spool/mail
```

12.5.33 Variable sys.nova_version

The variable gives the version of the running CFEngine Nova Edition.

```
# nova_version = 1.1.3
```

12.5.34 Variable sys.os

The name of the operating system according to the kernel.

```
# os = linux
```

12.5.35 Variable sys.ostype

Another name for the operating system.

```
# ostype = linux_x86_64
```

12.5.36 Variable sys.policy_hub

Hostname of the machine acting as a policy hub. This value is set during bootstrap. In case bootstrap was not performed, it is set to "undefined".

History: Was introduced in version 3.1.0b1,Nova 2.0.0b1 (2010). Available in Community since 3.2.0

```
reports:

 "Policy hub is $(sys.policy_hub)";
```

12.5.37 Variable sys.release

The kernel release of the operating system.

```
# release = 2.6.22.19-0.1-default
```

12.5.38 Variable sys.resolv

The location of the system resolver file.

```
# resolv = /etc/resolv.conf
```

12.5.39 Variable sys.uqhost

The unqualified name of the current host. See also `sys.fqhost`.

```
# uqhost = myhost
```

12.5.40 Variable sys.version

The version of the running kernel. On Linux, this corresponds to the ouput of `uname -v`.

```
# version = #55-Ubuntu SMP Mon Jan 10 23:42:43 UTC 2011
```

History: Was introduced in version 3.1.4,Nova 2.0.2 (2011)

12.5.41 Variable sys.windir

On the Windows version of CFEngine Nova, this is the path to the Windows directory of this system.

```
# windir = C:\WINDOWS
```

12.5.42 Variable sys.winprogdir

On the Windows version of CFEngine Nova, this is the path to the program files directory of the system.

```
# winprogdir = C:\Program Files
```

12.5.43 Variable sys.winprogdir86

On 64 bit Windows versions of CFEngine Nova, this is the path to the 32 bit (x86) program files directory of the system.

```
# winprogdir86 = C:\Program Files (x86)
```

12.5.44 Variable sys.winsysdir

On the Windows version of CFEngine Nova, this is the path to the Windows system directory.

```
# winsysdir = C:\WINDOWS\system32
```

12.5.45 Variable sys.workdir

The location of the CFEngine work directory and cache. For the system privileged user this is normally:

```
# workdir = /var/cfengine
```

For non-privileged users it is in the user's home directory:

```
# workdir = /home/user/.cfagent
```

On the Windows version of CFEngine Nova, it is normally under program files (the directory name may change with the language of Windows):

```
# workdir = C:\Program Files\CFEngine
```

12.6 Variable context `this`

This context '`this`' is used to access information about promises during their execution. It is context dependent and not universally meaningful or available, but provides a context for variables where one is needed (such as when passing the value of a list variable into a parameterized `edit_line` promise from a `file` promise). For example:

```
bundle agent resolver(s,n)
{
files:
  "$(sys.resolv)"

      create        => "true",
      edit_line     => doresolv("@(this.s)","@(this.n)"),
      edit_defaults => reconstruct;
}
```

Note that every unqualified variable is automatically considered to be in context '`this`', so that a reference to the variable `$(foo)` is identical to referencing `$(this.foo)`. You are strongly encourged to **not** take advantage of this behaviour, but simply to be aware that if you attempt to declare a variable name with one of the following special reserved names, CFEngine will issue a warning (and you can reference your variable by qualifying it with the bundle name in which it is declared).

12.6.1 Variable this.handle

This variable points to the promise handle of the currently handled promise; it is useful for referring to the intention in log messages.

12.6.2 Variable this.promise_filename

This variable reveals the name of the file in which the current promise is defined.

12.6.3 Variable this.promise_linenumber

This variable reveals the line number in the file at which it is used. It is useful to differentiate otherwise identical reports promises.

12.6.4 Variable this.promiser

The special variable `$(this.promiser)` is used to refer to the current value of the promiser itself, in a number of allowed cases, typically when searches can take place. Current promise types that define `$(this.promiser)` are: `files`, `processes`, `commands`.

This variable is useful in `files` promises, for instance when using pattern matching or `depth_search` that implicitly match multiple objects. In that case, `$(this.promiser)` refers to the currently identified file that makes the promise. For example:

```
bundle agent find666
{
files:
  "/home"
    file_select => world_writeable,
    transformer => "/bin/echo DETECTED $(this.promiser)",
    depth_search => recurse("inf");

  "/etc/.*"
    file_select => world_writeable,
    transformer => "/bin/echo DETECTED $(this.promiser)";
}

body file_select world_writeable
{
  search_mode => { "o+w" };
  file_result => "mode";
}
```

12.6.5 Variable service_policy

This variable is set to the values of the promise attribute `service_policy`, e.g.

```
services:

  "www"  service_policy => "start";
```

and is typically used in the adaptations for custom services bundles in the service methods, See Section 7.24.3 [service_method in services], page 360.

12.6.6 Variable this.this

From version core 3.3.0 this variables is reserved. It is used by functions like `maplist()` to represent the current object in a transformation map.

13 Logs and records

CFEngine writes numerous logs and records to its private workspace, referred to as 'WORKDIR'. This chapter makes some brief notes about these files. CFEngine approaches monitoring and reporting from the viewpoint of scalability so there is no default centralizatio of reporting information, as this is untenable for more than a few hundred hosts. Instead, in the classic CFEngine way, every host is reponsible for its own data. Solutions for centralization and netwide reporting will be given elsewhere.

The filenames referred to in this section are all relative to the CFEngine work directory 'WORKDIR'.

13.1 Embedded Databases

The embedded databases can be viewed and printed using the reporting tool `cf-report`.

Their file extensions will vary based on which library is used to implement them; either Tokyo Cabinet (`.tcdb`), Quick Database Manager (`.qdbm`), or Berkeley DB (`.db`). Converting one database format to another is not handled by CFEngine, but there exist external tools meant for that purpose.

'`cf_Audit.tcdb`'
: A compressed database of auditing information. This file grows very large is auditing is switched on. By default, only minor information about CFEngine runs are recorded. This file should be archived and deleted regularly to avoid choking the system.

'`cf_lastseen.tcdb`'
: A database of hosts that last contacted this host, or were contacted by this host which includes the times at which they were last observed.

'`cf_classes.tcdb`'
: A database of classes that have been defined on the current host, including their relative frequences, scaled like a probability.

'`checksum_digests.tcdb`'
: The database of hash values used in CFEngine's change management functions.

'`performance.tcdb`'
: A database of last, average and deviation times of jobs recorded by `cf-agent`. Most promises take an immeasurablely short time to check, but longer tasks such as command execution and file copying are measured by default. Other checks can be instrumented by setting a `measurement_class` in the `action` body of a promise.

'`stats.tcdb`'
: A database of external file attributes for change management functionality.

'`state/cf_lock.tcdb`'
: A database of active and inactive locks and their expiry times. Deleting this database will reset all lock protections in CFEngine.

'`state/history.tcdb`'
: Enterprise level versions of CFEngine maintain this long-term trend database.

'`state/cf_observations.tcdb`'
: This database contains the current state of the observational history of the host as recorded by `cf-monitord`.

'state/promise_compliance.tcdb'
: Enterprise CFEngine (Nova and above) database of individual promise compliance history. The database is approximate because promise references can change as policy is edited. It quickly approaches accuracy as a policy goes unchanged for more than a day.

'state/cf_state.tcdb'
: A database of persistent classes active on this current host.

'state/nova_measures.tcdb'
: Enterprise CFEngine (Nova and above) database of custom measurables.

'state/nova_static.tcdb'
: Enterprise CFEngine (Nova and above) database of static system discovery data.

13.2 Text logs

'promise_summary.log'
: A time-stamped log of the percentage fraction of promises kept after each run.

'cf3.HOSTNAME.runlog'
: A time-stamped log of when each lock was released. This shows the last time each individual promise was verified.

'cfagent.HOSTNAME.log'
: Although ambiguously named (for historical reasons) this log contains the current list of setuid/setgid programs observed on the system. CFEngine warns about new additions to this list. This log has been deprecated.

'cf_value.log'
: A time stamped log of the business value estimated from the execution of the automation system.

'cf_notkept.log'
: A list of promises, with handles and comments, that were not kept. Nova enterprise versions only.

'cf_repaired.log'
: A list of promises, with handles and comments, that were repaired. Nova enterprise versions only.

'reports/*'
: Enterprise versions of CFEngine use this directory as a default place for outputting reports.

'reports/class_notes'
: Class data in csv format for export to CMDB.

'state/file_change.log'
: A time-stamped log of which files have experienced content changes since the last observation, as determined by the hashing algorithms in CFEngine.

'state/vars.out'
: Enterprise level versions of CFEngine use this log to communicate variable data.

'state/*_measure.log'
: Enterprise level versions of CFEngine maintain user-defined logs based on specifically promised observations of the system.

13.3 Reports in outputs

The 'outputs' directory contains a time-stamped list of outputs generated by `cf-agent`. These are collected by `cf-execd` and are often E-mailed as reports. However, not all hosts have an E-mail capability or are online, so the reports are kept here. Reports are not tidied automatically, so you should delete these files after a time to avoid a build up.

13.4 Additional reports in commcerical CFEngine versions

13.5 State information

The CFEngine components keep their current process identifier number in 'pid files' in the work directory: e.g.

```
cf-execd.pid
cf-serverd.pid
```

Most other state data refer to the running condition of the host and are generated by `cf-monitord` (`cfenvd` in earlier versions of CFEngine).

'`state/env_data`'
: This file contains a list of currently discovered classes and variable values that characterize the anomaly alert environment. They are altered by the monitor daemon.

'`state/all_classes`'
: A list of all the classes that were defined the last time that CFEngine was run.

'`state/cf_*`'
: All files that begin with this prefix refer to cached data that were observed by the monitor daemon, and may be used by `cf-agent` in `reports` with `showstate`.

CFEngine®