Yodl 4.05.01

Frank B. Brokken (f.b.brokken@rug.nl)
initially by Karel Kubat
Center for Information Technology, University of Groningen

1996-NOW

Abstract

Yodl is a package implementing a pre-document language and tools to process it. The idea of Yodl is that you write up a document in a pre-language, then use the tools (e.g. yodl2html) to convert it to some final document language. Current converters are for HTML, man, LaTeX, a poor-man's text converter and an experimental XML converter. Main document types are 'article', 'report', 'book', 'letter' and 'manpage'. The Yodl document language is designed to be easy to use and extensible.

Contents

1 Introduction **14**

1.1 What's new since version 4.00.00? 15

1.2 Why use Yodl? 16

1.3 Copying Yodl 17

2 Yodl User Guide **18**

2.1 Using the yodl program 18

2.2 The Yodl grammar 21

2.2.1 Language elements 21

2.2.2 Line continuation 25

2.2.3 The +identifier sequence 26

2.2.4 Preventing macros from being expanded 27

2.3 Character tables 28

2.3.1 Defining character tables 28

2.3.2 Using character tables 29

2.3.3 Pushing and popping character tables 30

2.4 Sending literal text to the output 30

2.5 Counters 32

2.5.1 Creating counters 32

2.5.2 Using counters 33

3 All builtin functions **34**

3.1 Yodl's builtin commands 34

3.1.1 ADDTOCOUNTER 35

3.1.2 ADDTOSYMBOL 35

3.1.3 ATEXIT 35
3.1.4 CHAR 36
3.1.5 CHDIR 37
3.1.6 COMMENT 37
3.1.7 COUNTERVALUE 38
3.1.8 DECWSLEVEL 38
3.1.9 DEFINECHARTABLE 39
3.1.10 DEFINECOUNTER 39
3.1.11 DEFINEMACRO 40
3.1.12 DEFINESYMBOL 44
3.1.13 DELETECHARTABLE 45
3.1.14 DELETECOUNTER 45
3.1.15 DELETEMACRO 45
3.1.16 DELETENOUSERMACRO 45
3.1.17 DELETESYMBOL 46
3.1.18 ERROR 46
3.1.19 EVAL 46
3.1.20 FILENAME 48
3.1.21 FPUTS 48
3.1.22 IFBUILTIN 49
3.1.23 IFCHARTABLE 49
3.1.24 IFDEF 50
3.1.25 IFEMPTY 50
3.1.26 IFEQUAL 51
3.1.27 IFGREATER 52
3.1.28 IFMACRO 53
3.1.29 IFSMALLER 53
3.1.30 IFSTREQUAL 54
3.1.31 IFSTRSUB 55
3.1.32 IFSYMBOL 55
3.1.33 IFZERO 56

3.1.34 INCLUDEFILE 56
3.1.35 INCWSLEVEL 57
3.1.36 INTERNALINDEX 57
3.1.37 NOEXPAND 58
3.1.38 NOEXPANDINCLUDE 59
3.1.39 NOEXPANDPATHINCLUDE 60
3.1.40 NOTRANS 61
3.1.41 NOUSERMACRO 62
3.1.42 OUTBASE 62
3.1.43 OUTDIR 62
3.1.44 OUTFILENAME 62
3.1.45 PARAGRAPH 63
3.1.46 PIPETHROUGH 64
3.1.47 POPCHARTABLE 64
3.1.48 POPCOUNTER 64
3.1.49 POPMACRO 65
3.1.50 POPSUBST 65
3.1.51 POPSYMBOL 66
3.1.52 POPWSLEVEL 66
3.1.53 PUSHCHARTABLE 66
3.1.54 PUSHCOUNTER 67
3.1.55 PUSHMACRO 67
3.1.56 PUSHSUBST 68
3.1.57 PUSHSYMBOL 68
3.1.58 PUSHWSLEVEL 68
3.1.59 RENAMEMACRO 69
3.1.60 SETCOUNTER 69
3.1.61 SETSYMBOL 70
3.1.62 SUBST 70
3.1.63 SYMBOLVALUE 71
3.1.64 SYSTEM 72

3.1.65 TYPEOUT 72
3.1.66 UPPERCASE 72
3.1.67 USECHARTABLE 73
3.1.68 USECOUNTER 73
3.1.69 VERBOSITY 73
3.1.70 WARNING 75

4 Macros and Document types **76**

4.1 General structure of a Yodl document 77
4.1.1 Document types 78
4.1.2 The manpage document type 79
4.2 Predefined macros 82
4.2.1 abstract(text) 83
4.2.2 addntosymbol(symbol)(n)(text) 83
4.2.3 affiliation(site) 83
4.2.4 AfourEnlarged() 83
4.2.5 appendix() 83
4.2.6 article(title)(author)(date) 83
4.2.7 attrib(text) 83
4.2.8 attribclear() 84
4.2.9 attribinsert() 84
4.2.10 bf(text) 84
4.2.11 bind(text) 85
4.2.12 book(title)(author)(date) 85
4.2.13 cell(contents) 85
4.2.14 cells(nColumns)(contents) 85
4.2.15 cellsline(from)(count) 85
4.2.16 center(text) 86
4.2.17 chapter(title) 86
4.2.18 cindex() 86
4.2.19 cite(text) 86
4.2.20 clearpage() 86

4.2.21 code(text) . . . 86
4.2.22 columnline(from)(through) . . . 86
4.2.23 dashes() . . . 87
4.2.24 def(macroname)(nrofargs)(redefinition) . . . 87
4.2.25 description(list) . . . 87
4.2.26 dit(itemname) . . . 87
4.2.27 eit() . . . 87
4.2.28 ellipsis() . . . 87
4.2.29 em(text) . . . 87
4.2.30 email(address) . . . 88
4.2.31 enumeration(list) . . . 88
4.2.32 euro() . . . 88
4.2.33 evalsymbol(symbol)(expression) . . . 88
4.2.34 fig(label) . . . 88
4.2.35 figure(file)(caption)(label) . . . 88
4.2.36 file(text) . . . 89
4.2.37 findex() . . . 89
4.2.38 footnote(text) . . . 89
4.2.39 gagmacrowarning(name name ...) . . . 89
4.2.40 getaffilstring() . . . 89
4.2.41 getauthorstring() . . . 90
4.2.42 getchapterstring() . . . 90
4.2.43 getdatestring() . . . 90
4.2.44 getfigurestring() . . . 90
4.2.45 getpartstring() . . . 90
4.2.46 gettitlestring() . . . 90
4.2.47 gettocstring() . . . 90
4.2.48 htmlcommand(cmd) . . . 90
4.2.49 htmlheadfile(file) . . . 91
4.2.50 htmlheadopt(option) . . . 91
4.2.51 htmlnewfile() . . . 91

4.2.52 htmlstyle(tag)(definition) . . . 91
4.2.53 htmlstylesheet(url) . . . 92
4.2.54 htmltag(tagname)(start) . . . 92
4.2.55 ifnewparagraph(truelist)(falselist) . . . 92
4.2.56 includefile(file) . . . 93
4.2.57 includeverbatim(file) . . . 93
4.2.58 it() . . . 93
4.2.59 itdesc(itemname)(contents) . . . 93
4.2.60 itemization(list) . . . 93
4.2.61 kindex() . . . 93
4.2.62 label(labelname) . . . 94
4.2.63 langle() . . . 94
4.2.64 languagedutch() . . . 94
4.2.65 languageenglish() . . . 94
4.2.66 languageportugese() . . . 94
4.2.67 LaTeX() . . . 94
4.2.68 latexaddlayout(arg) . . . 94
4.2.69 latexcommand(cmd) . . . 94
4.2.70 latexdocumentclass(class) . . . 94
4.2.71 latexlayoutcmds(NOTRANSs) . . . 95
4.2.72 latexoptions(options) . . . 95
4.2.73 latexpackage(options)(name) . . . 95
4.2.74 lchapter(label)(title) . . . 95
4.2.75 letter(language)(date)(subject)(opening)(salutation)(author) 95
4.2.76 letteraddenda(type)(value) . . . 95
4.2.77 letteradmin(yourdate)(yourref) . . . 96
4.2.78 letterfootitem(name)(value) . . . 96
4.2.79 letterreplyto(name)(address)(zip city) . . . 96
4.2.80 letterto(element) . . . 96
4.2.81 link(description)(labelname) . . . 96
4.2.82 lref(description)(labelname) . . . 96

4.2.83 lsect(label)(title) . . . 96
4.2.84 lsubsect(label)(title) . . . 96
4.2.85 lsubsubsect(label)(title) . . . 97
4.2.86 lsubsubsubsect(label)(title) . . . 97
4.2.87 lurl(locator) . . . 97
4.2.88 mailto(address) . . . 97
4.2.89 makeindex() . . . 97
4.2.90 mancommand(cmd) . . . 97
4.2.91 manpage(title)(section)(date)(source)(manual) . . . 97
4.2.92 manpageauthor() . . . 97
4.2.93 manpagebugs() . . . 98
4.2.94 manpagedescription() . . . 98
4.2.95 manpagediagnostics() . . . 98
4.2.96 manpagefiles() . . . 98
4.2.97 manpagename(name)(short description) . . . 98
4.2.98 manpageoptions() . . . 98
4.2.99 manpagesection(SECTIONNAME) . . . 98
4.2.100 manpageseealso() . . . 98
4.2.101 manpagesynopsis() . . . 98
4.2.102 manttquoted(onOff) . . . 99
4.2.103 mbox() . . . 99
4.2.104 metaC(text) . . . 99
4.2.105 metaCOMMENT(text) . . . 99
4.2.106 mscommand(cmd) . . . 99
4.2.107 nbsp(count) . . . 99
4.2.108 nchapter(title) . . . 99
4.2.109 nemail(name)(address) . . . 99
4.2.110 nl() . . . 100
4.2.111 nodeprefix(text) . . . 100
4.2.112 nodeprefix(text) . . . 100
4.2.113 nodetext(text) . . . 100

4.2.114 nohtmlfive() . . . 100
4.2.115 nohtmlimgstyle() . . . 100
4.2.116 nop(text) . . . 100
4.2.117 nosloppyhfuzz() . . . 101
4.2.118 notableofcontents() . . . 101
4.2.119 notitleclearpage() . . . 101
4.2.120 notocclearpage() . . . 101
4.2.121 notransinclude(filename) . . . 101
4.2.122 noxlatin() . . . 101
4.2.123 nparagraph(title) . . . 101
4.2.124 npart(title) . . . 102
4.2.125 nsect(title) . . . 102
4.2.126 nsubsect(title) . . . 102
4.2.127 nsubsubsect(title) . . . 102
4.2.128 nsubsubsect(title) . . . 102
4.2.129 PageEnlarged(up)(height)(left)(width) . . . 102
4.2.130 paragraph(title) . . . 102
4.2.131 part(title) . . . 102
4.2.132 pindex() . . . 103
4.2.133 plainhtml(title) . . . 103
4.2.134 printindex() . . . 103
4.2.135 quote(text) . . . 103
4.2.136 rangle() . . . 103
4.2.137 redef(macro)(nrofargs)(redefinition) . . . 103
4.2.138 redefinemacro(macro)(nrofargs)(redefinition) . . . 103
4.2.139 ref(labelname) . . . 103
4.2.140 report(title)(author)(date) . . . 104
4.2.141 roffcmd(dotcmd)(sameline)(secondline)(thirdline) . . . 104
4.2.142 row(contents) . . . 104
4.2.143 rowline() . . . 104
4.2.144 sc(text) . . . 104

4.2.145 sect(title) . . . 105
4.2.146 setaffilstring(name) . . . 105
4.2.147 setauthorstring(name) . . . 105
4.2.148 setchapterstring(name) . . . 105
4.2.149 setdatestring(name) . . . 105
4.2.150 setfigureext(name) . . . 105
4.2.151 setfigurestring(name) . . . 105
4.2.152 sethtmlfigureext(ext) . . . 105
4.2.153 htmlmetacharset(meta-charset) . . . 106
4.2.154 setincludepath(name) . . . 106
4.2.155 setlanguage(name) . . . 106
4.2.156 setlatexalign(alignment) . . . 106
4.2.157 setlatexfigureext(ext) . . . 106
4.2.158 setlatexverbchar(char) . . . 106
4.2.159 setmanalign(alignment) . . . 107
4.2.160 setpartstring(name) . . . 107
4.2.161 setrofftab(x) . . . 107
4.2.162 setrofftableoptions(optionlist) . . . 107
4.2.163 settitlestring(name) . . . 108
4.2.164 settocstring(name) . . . 108
4.2.165 sgmlcommand(cmd) . . . 108
4.2.166 sgmltag(tag)(onoff) . . . 108
4.2.167 sloppyhfuzz(points) . . . 108
4.2.168 standardlayout() . . . 108
4.2.169 strong(contents) . . . 108
4.2.170 subs(text) . . . 109
4.2.171 subsect(title) . . . 109
4.2.172 subsubsect(title) . . . 109
4.2.173 subsubsubsect(title) . . . 109
4.2.174 sups(text) . . . 109
4.2.175 table(nColumns)(alignment)(Contents) . . . 109

4.2.176 tableatt(nColumns)(alignment)(Contents) 109
4.2.177 tac(alignment)(contents) 110
4.2.178 tao(type)(specification) 110
4.2.179 tbl(align)(contents) 111
4.2.180 tc(contents) 111
4.2.181 tcell(text) 112
4.2.182 telycommand(cmd) 112
4.2.183 TeX() 112
4.2.184 texinfocommand(cmd) 112
4.2.185 tindex() 112
4.2.186 titleclearpage() 113
4.2.187 tline(beginNr)(endNr) 113
4.2.188 tnac(nCells)(alignment)(contents) 113
4.2.189 tnc(nCells)(contents) 114
4.2.190 tocclearpage() 114
4.2.191 tr(contents) 114
4.2.192 tt(text) 115
4.2.193 ttbegin() 115
4.2.194 ttend() 116
4.2.195 twrap(value) 116
4.2.196 txtcommand(cmd) 116
4.2.197 url(description)(locator) 116
4.2.198 verb(text) 116
4.2.199 verbinclude(filename) 117
4.2.200 verbinsert(args) 117
4.2.201 verborg(text) 117
4.2.202 verbpipe(command)(text) 118
4.2.203 vindex() 118
4.2.204 whenhtml(text) 118
4.2.205 whenlatex(text) 118
4.2.206 whenman(text) 118

4.2.207 whenms(text) . . . 118
4.2.208 whensgml(text) . . . 118
4.2.209 whentely(text) . . . 118
4.2.210 whentexinfo(text) . . . 118
4.2.211 whentxt(text) . . . 119
4.2.212 whenxml(text) . . . 119
4.2.213 xit(itemname) . . . 119
4.2.214 xmlcommand(cmd) . . . 119
4.2.215 xmlmenu(order)(title)(menulist) . . . 119
4.2.216 xmlnewfile() . . . 119
4.2.217 xmlsetdocumentbase(name) . . . 120
4.2.218 xmltag(tag)(onoff) . . . 120
4.3 Conversion-related topics . . . 120
4.3.1 Conversion-type specific literal commands . . . 120
4.3.2 Figures . . . 122
4.3.3 Fonts and sizes . . . 124
4.3.4 Labels, links, references and URLs . . . 124
4.3.5 Lists and environments . . . 127
4.3.6 Sectioning . . . 130
4.3.7 Typesetting modifiers . . . 131
4.3.8 Miscellaneous commands . . . 133
4.4 Locations of the macros . . . 135

5 Conversions and converters 136

5.1 Conversion script invocations . . . 136
5.2 The HTML converter . . . 137
5.3 The LaTeX converter . . . 138
5.4 The man converter . . . 139
5.5 The txt converter . . . 139
5.6 The experimental XML converter . . . 140
5.7 The Yodl Post-processor ‘yodlpost’ . . . 140
5.8 The support program ‘yodlverbinsert’ . . . 141

5.8.1 Example 142

6 Technical information **144**

6.1 Obtaining Yodl 144

6.1.1 Installing Yodl 144

6.2 Organization of the software 146

6.2.1 Subdirectories and their meanings 146

6.3 Yodl's component interrelations and component setup 148

6.4 The token-producer 'lexer_lex()' 152

6.5 The Parser's Finite State Automaton 154

6.6 Adding a new macro 156

6.7 The Yodl post-processor 157

Chapter 1

Introduction

`Yodl` stands for 'Your Own Document Language' (originally: `Yet Oneother Document Language`) and is basically a pre-processor to convert document files in a special macro language (the `Yodl` language) to any output format. The `Yodl` language is not a 'final' language, in the sense that it can be viewed or printed directly. Rather, a document in the `Yodl` language is a 'pre-document', that is converted with some macro package to an output format, to be further processed.

`Yodl` was designed in 1996 by Karel Kubat when he needed a good document preprocessor to convert output to either LaTeX (for printing) or to HTML for publishing via a WWW site. Although SGML does this too, he wanted something that is used 'intuitively' and with greater ease. This is reflected in the syntax of the `Yodl` language, in the available macros of the `Yodl` macro package, and very probably also in other aspects of `Yodl`. However, `Yodl` is designed to convert to *any* output format; so it is possible to write a macro package that converts `Yodl` documents to, say, the `man` format for manual pages.

Some highlights of `Yodl`:

- `Yodl` allows the inclusion of files. This makes it easier to split up a document into 'logical' parts, each kept in a separate file. Thus, a 'main document' file can include all the sub-parts. (Imagine that you're the editor of a journal. Authors are likely to send in their submissions in separate files; inclusion can then be very handy!)
- Files which are included are searched for either 'as-is', or in a given 'system-wide include' directory, similar to the workings of the **C** preprocessor. Therefore, it is possible to create a set of include files holding macros, and to place them into one macro directory. (See also chapter 4, where a macro package that is distributed with `Yodl` is described.)
- For all the handled files (either stated on the commandline or included), `Yodl` supplies an extension if none is present. The default extension is `.yo`, but can be defined to anything in the compilation of the `Yodl` program.
- `Yodl` supports conditional parsing of its input, controlled by defined symbols. This resembles the `#ifdef` / `#else` / `#endif` preprocessor macros of the **C** language. `Yodl` also supports other `if` clauses, e.g., to test for the presence of an argument to a macro.

- `Yodl` offers hooks to define counters, to modify them, and to use them in a document. Thereby `Yodl` offers the possibility for automatic numbering of e.g., sections. Of course, some document languages (e.g., LaTeX) offer this too; but some don't. When converting a `Yodl` document to, say, HTML, this feature is very handy.
- `Yodl` is designed to be easy to use: `Yodl` uses 'normal' characters to identify commands in the text, instead of insisting weird-looking tags or escape characters. Editing a document in the `Yodl` macro language is designed to be as easy as possible.
- Similar to other document languages, `Yodl` supports 'character conversion tables' which define how a character should appear in the output.

This document first describes `Yodl` from the point of the user: how can macros be defined, how is the program used etc.. Next, my own macro package is presented and the macros therein described. Finally, this document holds technical information about the installation and the inner workings of `Yodl`.

1.1 What's new since version 4.00.00?

In `Yodl` Version 4.02.00 several new features were introduced.

- Three new builtin commands were implemented:

  STRLEN returns the length of its argument, or the length of a symbol's text if the symbol's name is passed as argument to `STRLEN`.

  SUBSTR returns a substring of a specified length, starting at a specified offset of its first argument (or returns a substring of a symbol's text if the symbol's name is passed as argument to `SUBSTR`).
- End-of-line comment is now available as the triplet `\//`. Following `\//` all subsequent characters on its line as well as initial blanks on the next line are skipped.
- Redesigned `yodlverbinsert`, which now also concatenates multiple identically marked sections.
- Added the macro, writing a non-breaking space character.
- Added the macro '`tbl`' (and support macros '`tr, tc, tnc, ta, tnac`' and '`tline`') as alternative to '`table`' (and support macros). The '`tbl`' macro extends '`table`', and simplifies defining tables.

In `Yodl` Version 4.00.00 some old features were removed, and several new ones were introducted.

- The following builtin commands, considered obsolete since `Yodl` 2.00.00, were removed:

  DUMMY no replacement.

  ENDDEF DECWSLEVEL should be used;

INCLUDELIT NOEXPANDINCLUDE should be used;

NEWCOUNTER DEFINECOUNTER should be used;

STARTDEF INCWSLEVEL should be used;

UNDEFINEMACRO DELETEMACRO should be used;

WRITEOUT FPUTS should be used;

- Three new builtin commands were implemented:

  PUSHSUBST allows temporary suppression (or activation) of `SUBST` definitions;

  POPSUBS restores the activation type of `SUBST` definitions that was active just before the last `PUSHSUBST` call;

  XXSUBST internally used builtin command.

- The `verb` macro now starts with `PUSHSUBST(0)` and ends with `POPSUBST()`.

- The following macros, considered obsolete since `Yodl` 2.00.00, were removed: `endcenter enddit endeit endit endmenu endtable itemize menu mit node startcenter startdit starteit startit startmenu starttable`

1.2 Why use Yodl?

Yodl is not a word processor, not even an editor. At first glance you might say, yeah, why should I learn Your Own Document Language? The answer is exactly that: because it can be *Your* own document language!

First of all, Yodl may lower the threshold of new users to start writing documents. An example of an excellent, though not very user-friendly document language is LaTeX. Typing all the backslash and curly brace characters in LaTeX and remembering that an asterisk must be typed as `$*$` may be hard at first. In such situations, a properly configured Yodl macro set removes these obstacles and thereby helps novices. Yodl is designed to be easy to learn. As the Yodl package is growing, so is the manual. The ease of 'learning Yodl' may thus somewhat diminish, but just keep in mind: as long as you need just plain texts, Yodl does OK. If you want more functionality, e.g., the composition of manual pages for Unix, dig into the documentation.

Second, Yodl permits to create more than one macro set, defining the same commands, but leading to different output actions. Thereby, the same input file can be converted to several output formats, depending on the loaded macro set. In this, Yodl is a 'general front' document language, which converts a Yodl document to a specialized language for further processing. This was of course one of my reasons to write Yodl: I needed a good converter for either LaTeX or HTML.

Third, Yodl always allows an 'escape route' to the output format. Most situations can be handled with Yodl macros, but sure enough, some users will want special actions for a given output format. A typical example for the necessity of such an escape route is the typesetting of mathematical formulas. Say you want to use Yodl for a document that is converted either to LaTeX (being a very good mathematical typesetter) or to HTML (a very poor mathematical typesetter). An approach might be to decide *inside the document* how to typeset a mathematical formula. Yodl provides conditional command processing to accomplish this. The decision would

be based on the output format: for LaTeX, you'd typeset the formula using all the facilities that LaTeX offers, and for HTML you'd use poor-mans typesetting. Typically, other pre-processors for documents don't allow such escape routes. Well, Yodl does.

1.3 Copying Yodl

Yodl is free software; it is distributed under the terms of the GNU General Public Licence. For details, please refer to the file COPYING.

Chapter 2

Yodl User Guide

This section describes the `yodl` program from the point of a meta-user, one who is interested in how macro files work, or one who wants to write a new converter. If you're just interested in using Yodl with the pre-existing converters and macro files, skip this chapter and continue with the macro package description (chapter 4).

The `Yodl` program the main converter of the Yodl package. The basic usage of the `yodl` program, `yodl`'s built-in macros, and the syntax of the Yodl language is described here.

2.1 Using the yodl program

`Yodl` reads one or more input files, interprets the commands therein, and writes one output file. The program is started as:

yodl *options inputfile [inputfile...]*

In this specification, the options are optional. Most options have 'long variants' also, which are mentioned in the following list. In this list, `-x, +NOTRANS(-{}-{})optionname` are two alternate ways to specify option x. If `-x` takes an argument, it may be specified immediately following the `-x`, but separating blanks may also be used. Options not taking arguments can be combined (e.g., `-a -b -c` may be combined to `-abc`). Arguments specified with long options should be separated from the long option using a `=` character.

The following options are currently available:

- `-D, +NOTRANS(-{}-{})define=NAME[=VALUE]`: Defines *name* as a symbol. This option is acts like `DEFINESYMBOL(NAME)()`. If `=VALUE` is added, `NAME` is initialized to `VALUE` (identically to `DEFINESYMBOL(NAME)(VALUE)`).
- `-d, +NOTRANS(-{}-{})definemacro=NAME=EXPANSION`: Defines `NAME` as macro expanding to `EXPANSION`
- `-h, +NOTRANS(-{}-{})help`: usage information is written to the standard error stream, describing all of Yodl's options.

- `-i, +NOTRANS(-{}-{})index[=file]`: 'file' is the name of the index file. By default <`outputbase`>`.idx` is used. No default when output is written to stdout. The index file is processed by Yodl's post-processor, `yodlpost`.

- `-I, +NOTRANS(-{}-{})include=DIR`: This defines the system-wide include directory where `Yodl` searches for its input files. E.g. a statement to include a given file, like:

  ```
  INCLUDEFILE(latex)
  ```

  Yodl now searches for the file `latex` in the current directory, and when that fails, in the system-wide include directory. The system-wide include directory is typically the place where the maintainer of a system stores macro-files for Yodl. This searching process applies to files that are included inside a document but also applies to filenames on the command line when invoking the `Yodl` program.

  The name of the included file (`latex` in the above example) is the bare name, `Yodl` supplies a default extension (`.yo`), if necessary.

  The `-I` option overrules Yodl's built-in name for the system-wide include directory. The built-in name is defined when compiling Yodl, and is, e.g., `/usr/share/yodl`. Furthermore, the definition may contain $HOME, which is replaced by the user's home directory if the 'home' or 'HOME' environment variable is defined. It may also contain $STD_INCLUDE, which is replaced by the compilation defined standard include path. The standard includepath may be overruled by either (in that order) the command line switch `-I` or the `tt(Yodl)_INCLUDE_PATH` environment variable. By default, the current directory is added to the standard include path. Hewver, when `-I` or `tt(Yodl)_INCLUDE_PATH` is used, the current directory must be mentioned explicitly. The individual directories need not be terminated by a /-character. In distributed `.deb` archives, the standard directory is defined as `/usr/share/yodl` (prefixed by the current working directory).

- `-k, +NOTRANS(-{}-{})keep-ws`: Since `Yodl` version 2.00 blanks at the begin and end of lines are ignored, even without a trailing \, when the 'white space level' is non-zero. Earlier versions kept these blanks. The legacy handling of white space at end of lines can by obtained using the `-k` flag. Note that white space are always kept when using verbatim copying, and when the white-space level is zero.

- `-m, +NOTRANS(-{}-{})messages=SET`: Set the so-called 'message level' to a combination of the SET `acdeinw`. The letters of this set have the following meanings:

  - `a`: alert. When an alert-error occurs, Yodl terminates. Here Yodl requests something of the system (like a `get_cwd()`), but the system fails.
  - `c`: critical. When a critical error occurs, Yodl terminates. The message itself can be suppressed, but exiting can't. A critical condition is, e.g., the omission of an open parenthesis at a location where a parenthesized argument should appear, or a non-existing file in an `INCLUDEFILE` specification (as this file should be parsed). A non-existing file with a `NOEXPANDINCLUDE` specification is a plain (non-critical) error.
  - `d`: debug. Probably too much info, like getting information about each character that was read by Yodl.

  - `e`: error. An error (like doubly defined symbols). Once an error has been encountered the remainder of the input is still parsed (up to a maximum number of errors), but no output is generated.
  - `i`: info. Not as detailed as 'debug', but still very much info, like information about media switches.
  - `n`: notice. Information about, e.g., calls to the builtin function calls.
  - `w`: warning. Something you should know about, but probably not affecting Yodl's proper functioning

  Non-configurable is the handling of an *emergency* message. These messages can't be suppressed, but shouldn't happen, as they point to some internal error. It would be appreciated to `receive information`[1] about these messages if they ever occur.

- `-n, +NOTRANS(-{}-{})max-nested-files=NR`: This option causes Yodl to abort when the number of nested input files exceeds `NR`, which is 20 by default. Exceeding this number usually means a circular definition somewhere in the document. This is the case when, a file `a.yo` includes `b.yo`, while `b.yo` includes `a.yo` etc.. It does not prevent recursive macro- or subst-replacements. For that the `-r` (`+NOTRANS(-{}-{})max-replacements`) option is available.

- `-o, +NOTRANS(-{}-{})output=FILE`: This option causes Yodl to write its output to `FILE`. By default, the output goes to the standard output stream. E.g., you can use `Yodl` to read a file `input` and to write to `output` with the following two commands:

  ```
  yodl input > output
  yodl -ooutput input
  ```

  The difference being that in the latter case an index file is generated, but not in the former case. Notice that writing an index file can be forced when the `+NOTRANS(-{}-{})index` option is specified.

- `-p, +NOTRANS(-{}-{})preload=CMD`: This option 'pre-loads' the string `cmd`. It acts as though `cmd` was the first command in the first input file that is processed by `Yodl`.

  More than one `+NOTRANS(-{}-{})preload=CMD` options may be present on the command line. Each of the commands is then processed in turn, before reading any file.

- `-r, +NOTRANS(-{}-{})max-replacements=NR`: This option causes Yodl to abort when the number of macro calls or subst-replacements exceeds `NR` $*$ `10,000`. By default, `NR` equals 1. Setting `+NOTRANS(-{}-{})max-replacements=0` implies that no macro- or subst-replacement checks are performed.

- `-t, +NOTRANS(-{}-{})trace`: This option enables tracing: while parsing, Yodl writes its output to the standard error stream. As is the case with the `-k` option, this option is defined for debugging purposes only.

- `-V, +NOTRANS(-{}-{})version`. This option shows `Yodl`'s actual version.

[1] mailto:f.b.brokken@rug.nl

- `-v, +NOTRANS(-{}-{})verbose`: This option increases Yodl's 'verbosity level' and may occur more than once. By default yodl shows alerting, critical, emergency and error messages. Each `+NOTRANS(-{}-{})verbose` option adds a next message level. In order, warning, notice, info and debug messages are added to this set. It is also possible to suppress messages. The `VERBOSITY` builtin can be used for that.
- `-W, +NOTRANS(-{}-{})warranty`. This option shows a warranty disclaimer and a copyright notice.
- `-w, +NOTRANS(-{}-{})warn`: The presence of this option caused Yodl to warn when, e.g., symbols are redefined.

The *inputfile* elements on the command line specify which files Yodl should process. All names are supplied with an extension[2]. The files are then searched for in the directories mentioned in the include-path. Files may also be specified using absolute pathnames.

Note that all filenames on the command line are input files. To define an output file, either use the `+NOTRANS(-{}-{})output` option or redirect the output.

2.2 The Yodl grammar

The grammar which is used by `Yodl` mixes 'real' text that should appear on the output with *markups*: commands for `Yodl`. The markups must follow a certain grammar, which is described in this section. `Yodl` therefore falls in the category of 'markup languages', in contrast to 'WYSIWYG'-programs. As a consequence, `Yodl` promotes concept-oriented writing.

Basically, Yodl only does 'something special' when it encounters the name of a builtin function or the name of a user-defined macro, followed by a parenthesized argument. Sometimes a function or macro requires multiple arguments, which must then be specified in sequence. All required parameter lists, however, must be specified within the same input file. It is not allowed to split the activation of a builtin function or macro over multiple input files. Plain text, on the other hand, may be split over multiple files.

In this section the elements of `Yodl`'s grammar are briefly discussed.

2.2.1 Language elements

At the lowest level, `Yodl`'s lexical scanner returns small pieces of information to its parser. These pieces of information are called *tokens*, and consist of elements like a blank space, a non-blank character, or an end-of-ile flag. These tokens are at too small an aggregation level to be useful for the current user-guide, so here we concentrate our discussion on the next aggregation level: compound elements and conceptual elements.

Compound elements relate to the basic tokens as words in a sentence to the individual letters of the words. These compound elements are identifiers, names, numbers, parameter lists and arguments.

[2]this extension is defined in the installation of Yodl and is usually `.yo`

Conceptual elements are found at the next higher aggregation levels: *builtin functions* are the buildin blocks for all of `Yodl`'s functionality, *symbols* and *counters* are `Yodl`'s *variables*, and (user defined) *macros* extend `Yodl`'s functionality beyond those of the basic builtin functions.

In the coming sections these basic and conceptual elements are discussed in greater detail.

Identifiers and Names

Identifiers are names that can have a special meaning in the `Yodl` language. E.g., the sequence `INCLUDEFILE` is an identifier: when followed by a filename in parentheses, `Yodl` will take some special action (in this case, read the file as a `Yodl`-source file).

Identifiers may consist of uppercase or lowercase characters. No other characters may appear in them.

In particular, *note* that this diverts from the well known definition for identifiers used in most programming languages: identifiers may not contain underscores, nor digits. `Yodl`, therefore, won't accept identifiers like `run4` or `under_score`.

Names are sequences of characters, not containing white space characters. (i.e., any series of characters not containing spaces, tabs or newlines). Names are allowed with certain builtin functions, like the `INCLUDEFILE` function, expecting the name of a file as its argument.

Numbers

Numbers consist of digits and an optional minus sign. They are most often used for so-called *counters*. In some contexts (e.g. with the builtin function VERBOSITY 3.1.69, *hexadecimal* numbers are allowed. Hexadecimal numbers have 16 'digits': the familiar 0-9, but also `a-f` (or `A-F`), representing the decimal values 10 until 15, respectively. Hexadecimal values are usually prefixed by `0x`, for example `0x4e`.

In other contexts (in particular, with character tables 2.3), octal numbers or character constants are allowed too.

An octal number only consists of the digits 0-7. In `Yodl`, octal values *must* consist of three digits, and *must* be preceded by a backslash.

Character constants may very well be considered numerical values. Character constants consist of a character value between single quotes, for example `'a'`.

Refer to section 2.3 for more detailed information about the use of octal values and character constants.

`Yodl` has no concept of floating point values nor does it have facilities for performing floating point arithmetic.

Arguments to builtin functions and macros

Arguments that are passed to `Yodl` builtin functions or user-defined macros must be surrounded by a pair of matching open- and close-parentheses. Each parameter

list contains exactly *one* argument.

An argument is recognized when it immediately follows the name of a builtin function or user-defined macro. Some functions or macros expect multiple arguments. In those cases, the required number of arguments must be provided, possibly separated from each other by white-space.

For example, the following shows how to call the builtin function `DEFINECOUNTER`, expecting two arguments:

```
DEFINECOUNTER(MyCounter)()
DEFINECOUNTER(MyCounter)    ()
DEFINECOUNTER(MyCounter)(12)
```

`Yodl` recognizes the arguments of macros when delimited by `(` and `)` characters. As long as the numbers of opening and closing parentheses match, `Yodl` correctly recognizes the list. E.g., given a hypothetical macro `somemacro`, the following code sample shows the macro followed by one argument:

```
somemacro(Here is a chunk of text.)
somemacro(Here is a some (more) text.)
```

An error occurs when the parentheses aren't balanced: i.e., when the argument consists of more opening than closing parentheses or *vice versa* To handle such situations, Yodl offers a 'literal-character' mechanism (see the `CHAR` macro in 3.1.4) and a 'global substitution' mechanism (see the `SUBST` macro in 3.1.62). For example, to send the text

```
here's a ")" closing parenthesis
```

as an argument to our hypothetical macro `somemacro`, the following can be used:

```
COMMENT(-- Alternative 1: using CHAR --)
somemacro(here's a "CHAR(41)" closing parenthesis)

COMMENT(-- Alternative 2: using SUBST --)
SUBST(closepar)(CHAR(41))
somemacro(here's a "closepar" closing parenthesis)
```

Both methods have disadvantages: the `CHAR` method requires you to remember that an ASCII 41 is a closing parenthesis. The `SUBST` method defines a string `closepar` that is *always* expanded to a closing parenthesis, wherever it may occur in the text. But whatever method is used, it should be clear by now that unbalanced arguments can be handled by Yodl. Also, remember that unbalanced parenthesis pairs are only relevant in argument lists. Yodl handles parentheses in normal text as ordinary characters.

Builtin functions

The building blocks of `Yodl`'s functionality are its *builtin functions*. Builtin functions exists to manipulate all of `Yodl`'s builtin types (character tables, counters, macros and symbols) and to do basic bookkeeping and flow-control: it is possible to test values of counters and symbols, to include other input files, to generate warning and error messages, and to start child- or subprocesses. Each builtin function is described in a separate subsection of section BUILTIN 3.1.

Character translation tables

Character translation tables exist to perform conversion specific transformations. For example, in `html`, the `\'e` is written as `é`, but in LaTeX it's written as `\'e`. Rather than using a potentially long if-else ladder to determine how to set a particular character, a character translation table can be used. The character translation table of a particular conversion is then activated only for that type of conversion.

Character table translations are used very late during the processing of `Yodl`'s input `s`: it is the output generator that handles the character translations. Consequently, macros or builtin function calls that might appear in a character's redefinition element of a character table are not expanded. In practice this never is a point of concern. In section 2.3 the use of character translation tables is discussed in detail.

Counters

Some document languages (notably LaTeX) automatically prefix numbers when typesetting sections, subsections, tables, figures etc.. Other document languages (e.g. `html`) don't.

Therefore, a macro package that converts a Yodl document to LaTeX doesn't need to provide the numbering of sections etc.. However, if you do want the numbering and if you want to convert documents to, say, `html`, then you must take care of the numbering yourself.

Counters exist to make this possible. Counters can be incremented, can be given a particular value, can be given a new value temporarily and can be removed. They always contain integral values, which may be negative.

Section 2.5 describes the use of counters in more detail.

Macros

Macros are comparable to builtin functions, but they can be defined in `Yodl` input files. Macros add functionality to `Yodl` exceeding the basic functionality of the builtin functions. Macros can have arguments, and they are used in exactly the same way as builtin functions are used.

When `Yodl` encounters a macro, it acts as follows:

- Its arguments are obtained, by reading its argument lists. These arguments

are not interpreted in any way. They are simply removed from the input, and stored for further processing;

- References to arguments in the macro's definition (using the `ARG#` notation, where `#` is the sequence number of a particular argument) are replaced by the literal text of the corresponding macro's arguments.
- The thus modified definition text is now pushed back into the input stream, to be processed by `Yodl`'s lexical scanner.

Defining macros is described in section 3.1.11. Macros may be defined, deleted, renamed, and temporarily given other definitions.

Nousermacros

When `Yodl` is started using the `-w` flag on the command line, then warnings are generated when Yodl encounters a possible macro name, followed by a parenthesized argument, without finding a macro by that name. Yodl then prints something like `cannot expand possible user macro`.

Examples of such sequences are, `The necessary file(s) are here`, or `see the manual page for sed(1)`. The candidate macros are `file` and `sed`, as these names could very well have been 'valid' user macros followed by their arguments.

A *nousermacro* can be defined to suppress these warnings, by informing `Yodl` that `file` and `sed` aren't macros. Nousermacros may be defined and undefined. See sections 3.1.41 and 3.1.16 for details).

Symbols

`Yodl` symbols contain text. They were introduced to allow the flexible expansion of text, the length and/or content of which cannot be determined in advance. In particular, symbols are useful to store a series of LaTeX document options, or a series of `html` body options. In earlier versions of `Yodl` complex and confusing constructions using nested definitions of macros were used for this. These macros were not only confusingly complex, but they also suffered from a hard-coded maximum. Symbols solve these drawbacks, and now that they are available, they are used for all natural situations in which an initially unknown piece of text must be stored. National language specific strings are another useful area in which symbols can be used. The symbol `CONTENTSHEADING` can be set to the name of the contents heading (e.g., `Contents` in English, `Inhoud` in Dutch, `Contenido` in Spanish, and macros can simply insert the value of the symbol `CONTENTSHEADING` at the appropriate location.

Symbols can be defined 3.1.12, removed 3.1.17, (temporarily 3.1.57 or permanently 3.1.61) be given another value; pushed symbol values can be restored 3.1.51 at a later point. Of course, their values can also be inserted 3.1.63 into `Yodl`'s output file.

2.2.2 Line continuation

To make the typing of input easier, `Yodl` allows you to end a line with a backslash character \ and to continue it on the next line. That way you can split long lines to

fit your screen. When processing its input, `Yodl` treats these lines as one long line, and ignores the final \ character. This feature only works when the \ character is the last one on the line (only a physical newline may follow).

When the line **following** the one with the \ character has leading spaces, then these are omitted. This allows you to 'indent' a file as you wish, while the space characters of the indentation are ignored by the `Yodl` program.

A trivial example is the following:

```
Grandpa and\
grandma are sitting on the sofa.
```

Due to the occurrence of the \ character in the sequence `and\`, `Yodl` combines the lines to

```
Grandpa andgrandma are sitting on the sofa.
```

Note that the spaces before `grandma` are ignored, since this is the second line following a \ character.

If you **do** want one or more spaces while joining lines with \, put the spaces **before** the \ character.
Summarizing:

- A Line ending in a backslash character is merged with the next line.
- This only happens if the \ character is the **last** character of the line, no spaces may appear behind the \.
- When merging lines, `Yodl` ignores leading spaces of the second line.

The question is of course, how do you accomplish that a line really ends with a \, when you do **not** want `Yodl` to merge it with the following line? In such a case, type a space character following your \: `Yodl` won't combine the lines. Or set the \ character as `CHAR(\)` or `CHAR(92)` (see section 3.1.4 for the `CHAR` macro).

When `Yodl` processes input files, and the white-space level exceeds zero (see section 3.1.35), then all lines are processed as if they terminated by a \. This behavior was implemented first with `Yodl` version 2.00. It can be suppressed using `Yodl`'s `-k` flag.

2.2.3 The +identifier sequence

There may be situations in which you must type a macro name right after a sequence of characters, while Yodl should recognize this. Imagine that someone wrote a great macro `footnote` for you[3], to typeset footnotes. If you'd type in a document:

```
The C Programming Languagefootnote(as defined by
```

[3]someone did, in fact, see the next chapter

```
Kernighan and Ritchie) ...
```

then of course Yodl would fail to see the start of a macro in the sequence `Languagefootnote`. You could say

```
The C Programming Language footnote(as defined by
Kernighan and Ritchie) ...
```

but that would introduce a space between `Language` and the footnote. Probably you don't want that, since spaces between a word and a footnote number look awful and because of the fact that the footnote number might be typeset on the following line.

For these special situations, Yodl recognizes the `+identifier` sequence as the start of a macro, while the `+` sign is effectively ignored. In the above example you could therefore use

```
The C Programming Language+footnote(as defined by
Kernighan and Ritchie) ...
```

The `+identifier` recognition only works when the identifier following the `+` sign is a macro. In all other situations, a `+` is just a plus-sign.

(The `+identifier` sequence furthermore plays an important role in macro packages. If you're interested, see the file `shared.yo` which is by default installed to `/usr/local/lib/yodl`.)

2.2.4 Preventing macros from being expanded

One more feature of the Yodl language remains to be described. In the previous section it was described how a macro may be called immediately following alphabetical characters. What about the opposite situation where we do *not* want a macro to be expanded in a particular situation? The `NOUSERMACRO` builtin command (cf. section 3.1.41) may be used to suppress the interpretation of a character sequence (e.g., `file(...)`) as a macro, but what if a macro should not be expanded in the occasional situation? For this case various solutions are available:

- First, the `tt(...)` and `verb(...)` macros may be used to suppress macro expansion. These macros also temporarily alter the typesetting font, though.
- Second, `NOEXPAND()` builtin command may be used: the macro name may be passed to `NOEXPAND()`, immediately followed by the 'argument list':

  ```
  Like this: NOEXPAND(NOEXPAND)(hello world)
  ```

- Third, the `nop()` macro may be used to separate a macro name from its argument list:

  ```
  Like this: NOEXPAND+nop()(hello world)
  ```

2.3 Character tables

The Yodl language provides a way to define character translation tables, to activate them, and to deactivate them. A character translation table defines how a character in the input will appear in the output.

There are two main reasons for the need of character translation tables. First, a document language becomes much easier to use when you can type an asterisk as * instead of `$*$` or `\verb/*/` (these are sequences from the LaTeX document language). Hence, a mechanism that expands a * in the input to to `\verb/*/` on the output, saves the users a lot of typing.

Second, forcing users to type weird sequences won't work if you're planning on converting the same Yodl document to a different output format. If the user types `\verb/*/` in the input to typeset an asterisk in the output, how should he or she arrive at a single * in the output in another output format?

The solution is of course to define the translation for an input character like * given the output format.

2.3.1 Defining character tables

The built-in macro `DEFINECHARTABLE` defines a character translation table. It takes two arguments: the name of the table and the character translations. Hence, each table is defined by its own name.

As an example of a table, consider the following fragment. It defines a table that translates the upper case characters `A` to `E` to their lower case equivalents:

```
DEFINECHARTABLE(tolower)(
    'A' = "a"
    'B' = "b"
    'C' = "c"
    'D' = "d"
    'E' = "e"
)
```

Each `DEFINECHARTABLE` statement **must** have a non-empty second parameter. "Empty" character tables cannot be defined, though one non-translation table is built-in.

The syntaxis of the second argument is as follows:

- On separate lines, input characters are mapped to a sequence to appear on the output.
- Per line, the input character is specified as `'c'`, `c` being any character. Escape-sequences from the **C** programming language can be used in this specification; Yodl supports the sequences `\a` (alert), `\b` (beep), `\f` (formfeed), `\n` (newline), `\r` (carriage return), `\t` (tab), and `\v` (vertical tab). Octal and hexadecimal constants may also be used. E.g., character `Y` may also be specified using the octal value `\131` or the hexadecimal value `\x59`. Any other character following a \ defines itself: `\\` represents a single backslash character.

- Following the character specification, a `=` must appear.
- Following that, a sequence of one or more characters appears, enclosed in double quotes, defining the translation. Again, escape sequences can be used, as in:

  ```
  '\n' = "End of line\n"
  ```

  Such a mapping adds the text `End of line` to each line, since each newline character in the input is replaced by the text `End of line`, followed by the newline itself.

  Starting with Yodl 2.14.0 octal and hexadecimal constants may also be used within the double quoted string. E.g., character `Y` may also be specified using the octal value `\131` or the hexadecimal value `\x59`. As an example where the octal/hexadecimal values may be useful consider the processing of a man-page. The character representations for the literal double quote (`"`) in `troff` is `\(dq\&`. However, since `(` cannot be written literally in the character translation table since that would result in unbalanced parentheses while processing the character table's definition. Also, `CHAR(40)` cannot be used, since character table conversiond are performed by the output generator, which is called after the macro expansions have been performed. This it would result in the literal text `CHAR`((40)) appearing in the manual page.

  Using the octal character representation in the chartable specification for the `"` character appearing in man-page the problem can now be solved. The actual specification used is:

  ```
  '"' = "\\\050dq\\&"
  ```

Translations which are **not** specified in the table are left to the default, which is to output the character as-is.

Note that the character table translation is something that the `yodl` program does as one of its last actions, just before sending text to the output file. The expansion text is not further processed by `yodl`, except for the conversion of **C**-type escape sequences to ordinary characters. The expansion text should therefore not be protected by, e.g., `NOTRANS` (unless of course you want some character to generate the text `NOTRANS` on the output).

2.3.2 Using character tables

A defined translation table is activated by the macro `USECHARTABLE`. This macro takes one argument, which may be:

- empty, in which case the default mapping is restored,
- a name of a previously defined character table.

The default mapping, selected when an empty argument is given, means that Yodl enters its 'zero translation state', meaning no character translation at all.

2.3.3 Pushing and popping character tables

Besides the previously described macro `USECHARTABLE()`, Yodl has one other mechanism of activating and deactivating character translation tables. This mechanism uses a stack, and hence, the related macros are appropriately named `PUSHCHARTABLE()` and `POPCHARTABLE()`.

- `PUSHCHARTABLE(name)` *pushes* the currently active translation table onto a stack, and activates the table identified by `name`. The argument may be empty; in that case, the zero-translation table is activated (analogously to `USECHARTABLE()`).
- `POPCHARTABLE()` activates the translation table that was last pushed. There is no argument to this macro.

Using the push/pop mechanism is handy when a table must be temporarily activated, but when it is not known which table exactly is active prior to the temporary activation. E.g., imagine that you need to use a character table called `listing` to typeset a listing, but that you do not know the current table. The pushing and popping mechanism is then used as follows:

```
COMMENT(First, we save the current table on the stack and
        we activate our "listing" table.)
PUSHCHARTABLE(listing)

COMMENT(Now the text is question is typeset.)
...

COMMENT(The previously active table is re-activated, whatever its name.)
POPCHARTABLE()
```

2.4 Sending literal text to the output

The Yodl program has several built-in macros that can be used to send literal text to the output file. The macros are listed in the chapter 3.1 and are briefly described here:

- `CHAR` takes one argument: the ASCII number of a character or the character itself. The character is sent to the output file without being translated with the currently active character translation table.
- `NOTRANS` takes one argument: the text in question. The text is neither parsed (i.e., macros in it are not expanded), nor translated with the current character translation table.
  The `NOTRANS` macro is conceptually like a series of `CHAR` macros.
- `NOEXPAND` takes one argument: the text in question. The text is not parsed, but it **is** translated with the current character translation table.
- `POPSUBST` takes no argument, and restores the `SUBST` interpretation status that was active just before the most recently called `PUSHSUBST` builtin macro call.

- PUSHSUBST takes one argument: 0 or a non-zero value (commonly 1). Following PUSHSUBSTS(0) SUBST-definitions are ignored; following PUSHSUBSTS(1) SUBST-definitions are used.

To illustrate the need for the distinction between NOTRANS and NOEXPAND, consider the following. The HTML converter (described in chapter 4) must be able to send HTML commands to the output file, but must also be able to send literal text (e.g., a source file listing). The HTML commands of course must be neither translated with any character table, nor must they be expanded in regard to macros. In contrast, a source file listing must be subject to character translations: the &, < and > characters can cause difficulties. Two possible macros for a HTML converter are:

```
COMMENT(--- htmlcommand(cmd) sends its argument as a HTML command
            to the output ---)
DEFINEMACRO(htmlcommand)(1)(NOTRANS(ARG1))

COMMENT(--- verb(listing) sends the listing to the output ---)
DEFINECHARTABLE(list)(
    '&'     =   "&amp;"
    '<'     =   "&lt;"
    '>'     =   "&gt;"
)

DEFINEMACRO(verb)(1)(
    USECHARTABLE(list)
    NOTRANS(<listing>)
    NOEXPAND(ARG1)
    NOTRANS(</listing>)
    USECHARTABLE(standard)
)
```

In this example it is assumed that a character translation table standard exists, defining the 'normal' translations. This table is re-activated in the verb macro.

The currently defined verb macro for, e.g., html conversion looks like this:

```
DEFINEMACRO(verb)(1)(
    PUSHSUBST(0)
    NOTRANS(<pre>)
    XXnl()
    NOEXPAND(ARG1)
    XXnl()
    NOTRANS(</pre>)
    XXnl()
    POPSUBST()
)
```

Here, the surrounding PUSHSUBST(0) and POPSUBST() pair prevent the interpretation of multi-char SUBST definitions like

```
SUBST(\"e)(+NOTRANS(&euml;))
```

inside `verb` environments, causing unexpected translations in pieces of code like

```
cout << "\"evil\" code\n";
```

which, if the `PUSHSUBST` command isn't used, is converted to

```
cout << "&eacute;vil\" code\n";
```

2.5 Counters

Some document languages (notably LaTeX) automatically prefix numbers when typesetting sections, subsections, tables, figures etc.. Other document languages (e.g. HTML) unfortunately don't.

Therefore, a macro package that converts a Yodl document to LaTeX doesn't need to provide the numbering of sections etc.. However, if you do want the numbering and if you want to convert documents to, say, HTML, then you must take care of the numbering yourself.

This section describes the counters in Yodl: how to create counters, how to use them, etc..

2.5.1 Creating counters

Before a counter can be used, it must be created using `DEFINECOUNTER` or `PUSHCOUNTER`. These functions expects two arguments: the name of the counter and an (optional) integral additive expression. When provided, the expression may contain binary + and unary or binary - operators; its operands may be integral values or the names of existing counters. Additive expressions may not contain white space characters. If an additive expression is provided it is evaluated and its value is then used to initialize the named counter. If the second argument is empty then the counter is initialized to zero (0).

For example, let's say that our macro package should provide two sectioning commands: `section` and `subsection`. The sections should be numbered 0, 1, 2, etc., and the subsections 1.1, 1.2, 1.3 etc.. For this we can define two counters:

```
DEFINECOUNTER(sectcounter)()\//       initialized to 0
DEFINECOUNTER(subsectcounter)(1)\//   initialized to 1
```

2.5.2 Using counters

The builtin function `COUNTERVALUE(somecounter)` expands to the value of `somecounter`. E.g., if the current value is 2, then the value 2 is inserted into the output object. It is an error to use `COUNTERVALUE` on a non-existing counter or on a counter not having a defined value (see below).

Yodl has several functions to modify and/or to set the values of counters. The counter's value, named `expression` below, is an additive expression: it may contain binary + and unary or binary - operators. The operands may be integral values of the names of existing counters. Additive expressions may not contain blank spaces.

The functions modifying values of counters are:

- `POPCOUNTER(somecounter)`: This function pops the most recently pushed value off the counter's stack, assigning it to `somecounter`. An error occurs when `somecounter` doesn't exist. If the counter was never pushed, it still exists following `POPCOUNTER`, but its value is undefined: using `COUNTERVALUE(somecounter)` in that case generates an error.
- `PUSHCOUNTER(somecounter)(expression)`: This function pushes the current value of the counter `somecounter` on the counter's stack, making `expression's` value its new value. Its second argument may be empty in which case the counter is initialized to 0. When `somecounter` doesn't yet exist, it is created with an initial value of `expression`.
- `SETCOUNTER(somecounter)(expression)`: This function sets the value of `somecounter` to `expression's` value. The function does not expand to anything; i.e., it does not write to the output file.
- `ADDTOCOUNTER(somecounter)(expression)`: This function adds `expression's` value to `somecounter`.
- `USECOUNTER(somecounter)`: This function first increments `somecounter's` value, and then writes the value of the counter to the output file.

  This function is particularly useful in combination with `DEFINECOUNTER`: since `DEFINECOUNTER` initializes a counter to zero, `USECOUNTER` can be used to increment the value and to output it. The first time that `USECOUNTER` is used on a new counter, the value 1 appears on the output file. The next time, expression 2 appears on the output file etc..

Given the numbering requirements of the hypothetical commands `section` and `subsection` (see the previous section), we can now complete the definitions:

```
DEFINECOUNTER(sectcounter)
DEFINECOUNTER(subsectcounter)

DEFINEMACRO(section)(1)(\
SETCOUNTER(subsectcounter)(0)\
USECOUNTER(sectcounter) ARG1)

DEFINEMACRO(subsection)(1)(\
COUNTERVALUE(sectcounter).USECOUNTER(subsectcounter) ARG1)
```

Chapter 3

All builtin functions

3.1 Yodl's builtin commands

As mentioned previously, `Yodl`'s input consists of text and of commands. `Yodl` supports a number of built-in commands which may either be used in a `Yodl` document, or which can be used to create a macro package.

Don't despair if you find that the description of this section is too technical. Exactly for this reason, `Yodl` supports the macro packages to make the life of a documentation writer easier. E.g., see chapter 4 that describes a macro package for `Yodl`.

Most built-in functions and macros expand the information they receive the way they receive the information. I.e., the information itself is only evaluated by the time it is eventually inserted into an output medium (usually a file). However, some builtin functions *evaluate* their argument(s) once the argument is processed. They are:

- The `ERROR` built-in function (see section 3.1.18);
- The `EVAL` built-in function (see section 3.1.19);
- The `FPUTS` built-in function (see section 3.1.21);
- The `INTERNALINDEX` built-in function (see section 3.1.36);
- The `PUSHSUBST` built-in function (see section 3.1.56);
- The `TYPEOUT` built-in function (see section 3.1.65);
- The `UPPERCASE` built-in function (see section 3.1.66);
- The `WARNING` built-in function (see section 3.1.70);
- The `XXSUBST` internal use only built-in function;

All other built-in functions will *not* evaluate their arguments. See the mentioned functions for details, and in particular `EVAL()` for a description of this evaluation process.

3.1.1 ADDTOCOUNTER

The `ADDTOCOUNTER` function adds a given value to a counter. It expects two arguments: the counter name, and an additive expression defining the value to add. The counter must be previously created with `DEFINECOUNTER`.

The additive expression may not contain blank spaces and may use + and - operators, its operands may either be integral numeric values or names of (defined) counters. The resulting value can be negative; in that case, a value is subtracted from the destination counter.

For example, if `one` and `two` are counters, then

```
ADDTOCOUNTER(one)(-two)\//    subtracts two's value from one
ADDTOCOUNTER(one)(two+two)\// adds 2 x two's value to one
```

See further section 2.5.

3.1.2 ADDTOSYMBOL

Since Yodl version 2.00 symbols can be manipulated. To add text to an existing symbol the builtin `ADDTOSYMBOL` is available. It expects two parameter lists: the symbol's name, and the text to add to the symbol. The symbol must have been created earlier using DEFINECOUNTER (see section 3.1.10). The macro's second argument is not evaluated while `ADDTOSYMBOL` is processed. Therefore, it is easy to add the text of another symbol or the expansion of a macro to a symbol value. E.g.,

```
ADDTOSYMBOL(one)(SYMBOLVALUE(two)XXnl())
```

This adds the text of symbol `two`, followed by a new line, to the contents of symbol `one` only when symbol `one` is evaluated, not when `ADDTOSYMBOL` is evaluated.

Example:

```
ADDTOSYMBOL(LOCATION)(this is appended to LOCATION)
```

3.1.3 ATEXIT

`ATEXIT` expects one argument. The argument is appended to the output file. Note that this text is subject to character table translations etc..

An example using this function is the following. A document in the LaTeX typesetting language requires `\end{document}` to occur at the end of the document. To

automatically append this string to the output file, the following specification can be used:

```
ATEXIT(NOEXPAND(\end{document}))
```

Several ATEXIT lists can be defined. They are appended to the output file in the **reverse** order of specification; i.e., the first ATEXIT list is appended to the output file last. That means that in general the ATEXIT text should be specified when a 'matching' starting command is sent to the output file; as in:

```
COMMENT(Start the LaTeX document.)
NOEXPAND(\begin{document})

COMMENT(Ensure its proper ending.)
ATEXIT(NOEXPAND(\end{document}))
```

3.1.4 CHAR

The command CHAR takes one argument, a number or a character, and outputs its corresponding ASCII character to the final output file. This command is built for 'emergency situations', where you need to typeset a character despite the fact that it may be redefined in the current character table (for a discussion of character tables, see 2.3). Also, the CHAR function can be used to circumvent Yodl's requirement that open- and close-parentheses must match.

The following arguments may be specified with CHAR (attempted in this order):

- A decimal number indicating the number of the character in the ascii-table (for example CHAR(41));
- A plain, single character (for example CHAR(#)).

So, when you're sure that you want to send a printable character that is not a closing parenthesis to the output file, you can use the form CHAR(c), c being the character (as in, CHAR(;)). To send a non-printable character or a closing parenthesis to the output file, look up the ASCII number of the character, and supply that number as argument to the CHAR command.

Example: The following two statements send an A to the output file.

```
CHAR(65)
CHAR(A)
```

The following statement sends a closing parenthesis:

```
CHAR(41)
```

Another way to send a string to the output file without expansion by character tables or by macro interpretation, is by using the function `NOTRANS` (see section 3.1.40). If you want to send a string to the output **without** macro interpretation, but **with** character table translation, use `NOEXPAND` (see section 3.1.37).

3.1.5 CHDIR

The command `CHDIR` takes one argument, a directory to change to. This command is implemented to simplify the working with `includefile` (see `includefile` in `yodlmacros(7)`). As a demonstration, consider the following fragment:

```
includefile(subdir/onefile)
includefile(subdir/anotherfile)
includefile(subdir/yetanotherfile)
```

This fragment can be changed to:

```
CHDIR(subdir)
includefile(onefile)
includefile(anotherfile)
includefile(yetanotherfile)
CHDIR(..)
```

The current directory, as given to `CHDIR`, only affects how `includefile` searches for its files.

Note that this example assumes that the current working directory is a member of Yodl's include-path specification (cf., Yodl's `+NOTRANS(-{}-{})include` option).

3.1.6 COMMENT

The `COMMENT` function defines one parameter list. The text that is passed as argument is treated as comment. I.e., it is ignored; it is not copied to the final output file.

As an alternative to (short) `COMMENT` the triplet `\//` can be used. It starts 'end of line' comment, ignoring all characters on a line starting at `\//` up to the first non-blank character encountered on the next line. If the next line's first non-blank characters are `\//`, then that begins another end of line comment, which will therefore also be skipped.

To actually write `\//` or, using the current font: \// in a yodl-converted document, write, e.g., `tt(\)tt(//)` or, using the current font: nop(/)// in a yodl-source file, and write `\CHAR(/)/` in `verb` sections.

Example:

```
Hello world\// producess Hello world, skipping the rest
    \// this line is completely ignored
   s\// at this point Hello worlds has been produced.
```

3.1.7 COUNTERVALUE

`COUNTERVALUE's` argument expands to the value of a counter. Its single argument must contain the name of a counter. The counter must have been created earlier using the builtin `DEFINECOUNTER`.
Example:

```
The counter has value COUNTERVALUE(MYCOUNTER).
```

See also section 2.5.

3.1.8 DECWSLEVEL

`DECWSLEVEL` requires one (empty) argument. It reduces the current white-space level. The white-space level typically is used in files that only define Yodl macros. When no output should be generated while processing these files, the white-space level can be used to check for this. If the white-space level exceeds zero, a warning is generated if the file produces non-whitespace output. The builtin function `DECWSLEVEL` is used to reduce the whitespace level following a previous call of `INCWSLEVEL`.

Once the white space level exceeds zero, no output will be generated. White space, therefore effectively is ignored. The white space level cannot be reduced to negative values. A warning is issued if that would have happened if it were allowed.

Example:

```
INCWSLEVEL()
DEFINESYMBOL(....)
DEFINEMACRO(...)(...)(...)
DECWSLEVEL()
```

Without the `INCWSLEVEL` and `DECWSLEVEL`, calls, the above definition would generate four empty lines to the output stream.

The `INCWSLEVEL` and `DECWSLEVEL` calls may be nested. The best approach is to put an `INCWSLEVEL` at the first line of a macro-defining Yodl-file, and a matching `DECWSLEVEL` call at the very last line.

3.1.9 DEFINECHARTABLE

DEFINECHARTABLE is used to define a character translation table. The function expects two parameterlists, containing the name of the character table and character table translations on separate lines. These character table translations are of the form

```
character = quoted-string
```

Here, character is always a value within single quotes. It may be a single character, an octal character value or a hexadecimal character value. The single character may be prefixed by a \-character (e.g., '\\'). The octal character value must start with a backslash, followed by three octal digits (e.g., '\045'. The hexadecimal character value starts with 0x, followed by two hexadecimal characters. E.g., '0xbe'. The double quoted string may contain anything (but the string must be on one line), possibly containing escape-sequences as well: in the double quoted string the standard **C** escape sequences \a (alert), \b (beep), \f (formfeed), \n (newline), \r (carriage return), \t (tab), and \v (vertical tab) are recognized and automatically converted to their special meanings. Starting with Yodl 2.14.0 octal and hexadecimal constants may also be used. E.g., character Y may also be specified using the octal value \131 or the hexadecimal value \x59. Any other character following a backslash character (\) defines itself: \\ represents a single backslash character.

Example:

```
DEFINECHARTABLE(demotable)(
    '&'     = "&amp;"
    '\\'    = "\\backslash"
    '\045'  = "oct(45)"
    '0xa4'  = "hex(a4)"
)
```

The builtin function DEFINECHARTABLE does not *activate* the table. The table is merely defined. To activate the character translation table, use USECHARTABLE. The discussion of character tables is postponed to section 2.3.

3.1.10 DEFINECOUNTER

DEFINECOUNTER creates a new counter. This builtin function expects two arguments: the name of the counter and an additive expression whose value is used to initialize the counter.

The additive expression may not contain blank spaces and may use + and - operators, its operands may either be integral numeric values or names of (defined) counters. The resulting value can be negative; in that case, a value is subtracted from the destination counter.

Examples:

```
DEFINECOUNTER(year)(1950)
DEFINECOUNTER(nTimes)(year+12)\// initializes nTimes to 1962
```

See also section 2.5 and the `USECOUNTER` and `ADDTOCOUNTER` builtin functions.

3.1.11 DEFINEMACRO

`DEFINEMACRO` is used to define new macros. This function expects three arguments:

- An identifier, being the name of the macro to define. This identifier may only consist of uppercase or lowercase characters. Note that it can *not* contain numbers, nor underscore characters.
- A number, stating the number of arguments that the macro will require once it's used. The number must be in the range 0 to 61.
- The text that the macro expands to, once used. This text may contain the strings `ARG`x, x being 1, 2, etc.. At these places the arguments to the macro are pasted in. The numbers that identify the arguments are 1 to 9, then A to Z and finally a to z. This gives a range of 61 expandable arguments, which is enough for all real-life applications.

For example, the following fragment defines a macro `bookref`, which can be used to typeset a reference to a book. It requires three arguments; say, an author, a title and the name of a publisher:

```
DEFINEMACRO(bookref)(3)(
    Author(s):          ARG1
    Book title:         ARG2
    Published by:       ARG3
)
```

Such a macro could be used as follows:

```
bookref(Sobotta/Becher)
       (Atlas der Anatomie des Menschen)
       (Urban und Schwarzenberg, Berlin, 1972)
```

When called, it would produce the following output:

```
    Author(s):          Sobotta/Becher
    Book title:         Atlas der Anatomie des Menschen
    Published by:       Urban und Schwarzenberg, Berlin, 1972
```

While applying a macro, the values of the three arguments are pasted to the places where `ARG1`, `ARG2` etc. occur in the definition.

Note the following when defining new macros:

- The argument containing the name of the new macro, `(bookref)` in the above example, must occur right after `DEFINEMACRO`. No spaces are allowed in between. Space characters and newlines may however occur following this first argument.

  This behavior of the `yodl` program is similar to the usage of the defined macro: the author information must, enclosed in parentheses, follow right after the `bookref` identifier. I implemented this feature to improve the distinguishing between macros and real text. E.g., a macro `me` might be defined, but the text

```
I like me (but so do you)
```

  still is simple text; the macro `me` only is activated when a parenthesis immediately follows it.

- Be careful when placing newlines or spaces in the definition of a new macro. E.g., the definition, as given:

```
DEFINEMACRO(bookref)(3)(
    Author(s):          ARG1
    Book title:         ARG2
    Published by:       ARG3
)
```

  introduces extra newlines at the beginning and ending of the macro, which are copied to the output each time the macro is used. The extra newline occurs, of course, right before the sequence `Author(s):` and following the evaluation of `ARG3`. A simple backslash character at the end of the `DEFINEMACRO` line would prevent the insertion of extra newline characters:

```
DEFINEMACRO(bookref)(3)(\
    Author(s):          ARG1
    Book title:         ARG2
    Published by:       ARG3
)
```

- Note that when a macro is used which requires no arguments at all, one empty argument still must be specified. E.g., my macro package (see chapter 4) defines a macro `it` that starts a bullet item in a list. The macro takes no arguments, but still must be typed as `it()`.

  This behavior is consistent: it helps distinguish which identifiers are macros and which are simple text.

- Macro arguments may evaluate to text. When a \ is appended to the macro-argument, or in the default input handling within a non-zero white-space level (see section 3.1.35) this may invalidate a subsequent macro call. E.g., the macro

```
DEFINEMACRO(oops)(1)(
    ARG1
    XXnl()
)
```

when called as `oops(hello world)`, produces the output:

```
hello worldXXnl()
```

To prevent this gluing to arguments to subsequent macros, a single + should be prepended to the macro call:

```
DEFINEMACRO(oops)(1)(
    ARG1
    +XXnl()
)
```

See also section 2.2.3 obout the '+identifier'-sequence.

- Note the preferred layout of macro definitions and macro calls. Adhere to this form, to prevent drowning in too many parentheses. In particular:
  - Put all elements of the macro definition on one line, except for the macro-expansion itself. Each expansion element should be on a line by itself.
  - When calling macros put the macro's arguments underneath each other. If the macrolists themselves contain macro-calls, put each call again on a line of its own, indenting one tab-position beyond the location of the opening parenthesis of the argument.
  - No continnuation backslashes are required between arguments.
  - With complex calls, indent just the arguments, and put the parentheses in their required of logical locations.

Example of a complex call:

```
complex(
    first(
        ARG1
    )(
        ARG2
        +XXnl()
    )
    ARG3
```

```
        +nop()
        ARG4
        +XXnl()
    )
```

- Macro expansion proceeds as follows:
  - The arguments are read from the input
  - The contents of the arguments then replace their `ARGx` references in the macro's definition (in some exceptional cases, clearly indicated as such when applicable, the arguments themselves are evaluated first, and then these evaluated arguments are used as replacements for their corresponding `ARGx` references).
  - The now modified macro is read by Yodl's lexical scanner. This may result in yet another macro expansion, which will then be evaluated recursively.
  - Eventually, all expansion is completed (well, should complete, since Yodl doesn't test for eternal recursion) and scanning of the input continues beyond the original macro call.

  For example, assume we have the following two macros:

```
DEFINEMACRO(First)(1)(
    Hello ARG1
    +XXnl()
)
DEFINEMACRO(Second)(1)(
    First(ARG1)
    First(ARG1)
)
```

  and the following call is issued:

```
Second(Yodl)
```

  then the following happens:

  - `Second(Yodl)` is read as encountered.
  - `ARG1` in `Second` is replaced by `Yodl`, and the resulting macro body is sent to the lexical scanner for evaluation: It will see:

```
First(Yodl)First(Yodl)
```

  - The first call to `First()` is now evaluated. This puts (after replacing `ARG1` by `Yodl`) the following on the scanner's input:

```
Hello Yodl+XXnl()First(Yodl)
```

– `Hello Yodl` contains no macro call, so it is written to the output stream. Remains:

```
+XXnl()First(Yodl)
```

– Assume `XXnl()` merely contains a newline (represented by `\n`, here), so `+XXnl()` is now replaced by `\n`. This results in the following input for the lexical scanner:

```
\nFirst(Yodl)
```

– The `\n` is now written to the output stream, and the scanner sees:

```
First(Yodl)
```

– The second call to `First()` is now evaluated. This puts the following on the scanner's input:

```
Hello Yodl+XXnl()
```

– `Hello Yodl` is written to the output stream. Remains:

```
+XXnl()
```

– `+XXnl()` is now replaced by `\n`. The lexical scanner sees:

```
\n
```

– The newline is printed and we're done.

3.1.12 DEFINESYMBOL

`DEFINESYMBOL` expects two arguments. An identifier, which is the name of the symbol to define, and the textual value of the symbol. If the second argument is empty, the symbol is defined, but has an empty value.

The earlier interpretation of a Yodl symbol as a logical flag can still be used, but allowing it to obtain textual values greatly simplifies various Yodl macros.

Example:

```
DEFINESYMBOL(Yodl)(Your own document language)
DEFINESYMBOL(Options)()
```

3.1.13 DELETECHARTABLE

`DELETECHARTABLE` removes a definition of a character table that was defined by `DEFINECHARTABLE`. This function expects one argument: the name of the character table remove.

It's an error to attempt to delete a character table that is currently in use or to attempt to delete a non-existing character table.

Example:

```
DELETECHARTABLE(mytable)
```

3.1.14 DELETECOUNTER

`DELETECOUNTER` removes a definition of a counter that was defined by `DEFINECOUNTER`. This function expects one argument: the name of the counter to remove.

If the counter does not exist, a warning is issued. It is not considered an error to try to delete a counter that has not been defined earlier.

Example:

```
DELETECOUNTER(mycounter)
```

3.1.15 DELETEMACRO

`DELETEMACRO` removes a definition of a macro that was defined by `DEFINEMACRO`. This function takes one argument: the macro name to remove.

There is no error condition (except for syntax errors): when no macro with a matching name was previously defined, no action is taken.

For example, the safe way to define a macro is by first undefining it. This ensures that possible previous definitions are removed first:

Example:

```
DELETEMACRO(mymacro)
```

3.1.16 DELETENOUSERMACRO

`DELETENOUSERMACRO` removes a 'nousermacro' definition. The function expects one argument: the name of the 'nousermacro' identifier to be removed from the nousermacro-set.

There is no error condition (except for syntax errors): when the identifier wasn't stored as a 'nousermacro' no action is taken.

Example:

```
DELETENOUSERMACRO(mymacro)
```

3.1.17 DELETESYMBOL

`DELETESYMBOL` removes the definition of a symbol variable. It expects one argument, holding the name of the variable to deleted.

This macro has no error condition (except for syntax errors): the symbol in question may be previously defined, but that is not necessary.

Example:

```
DELETESYMBOL(Options)
```

3.1.18 ERROR

The `ERROR` function takes one argument: text to display to the standard error stream. The current input file and line number are also displayed. After displaying the text, the `yodl` program aborts with an exit status of 1.

The text passed to the function is expanded first. See the example.

The `ERROR` function is an example of a function that evaluates its argument itself.

This command can be used, e.g., in a macro package when an incorrect macro is expanded. In my macro package (see chapter 4) the `ERROR` function is used when the sectioning command `chapter()` is used in an `article` document (in the package, `chapter`'s are only available in `book`s or `report`s).

An analogous builtin function is `WARNING`, which also prints a message but does not exit (see section 3.1.70).

Example: In the following call, `COUNTERVALUE(NTRIES)` is replaced by its actual value:

```
ERROR(Stopping after COUNTERVALUE(NTRIES) attempts)
```

3.1.19 EVAL

The `EVAL` function takes one argument: the text to be evaluated. This function allows you to perform an indirect evaluation of Yodl commands. Assume that there

is a symbol `varnam` containing the name of a counter variable, then the following displays the counter's value, after having incremented it:

```
EVAL(NOTRANS(USECOUNTER)(SYMBOLVALUE(varnam)))
```

Here, `EVAL` performs the following steps:

- First, `NOTRANS(USECOUNTER)` is evaluated, producing `USECOUNTER`.
- Next, the open parenthesis is processed, producing the open parenthesis itself
- Then, `SYMBOLVALUE(varnam)` is evaluated, producing the name of a counter, e.g. '`counter`'.
- The closing parentheis is processed, producing the closing parenthesis itself.
- All this results in the *text*

  ```
  USECOUNTER(counter)
  ```

- This text is presented to Yodl's lexical scanner, resulting in incrementing the counter, and displaying its incremented value.

b(Caveat): macro arguments themselves are usually not evaluated. So, a construction like

```
USECOUNTER(EVAL(SYMBOLVALUE(varnam)))
```

fails, as '`EVAL(SYMBOLVALUE(varnam))`' is not a legal name for a counter. Here the `EVAL()` call is used as an argument, and is therefore not expanded.

The distinction is subtle, and is a consequence of the fact that builtin functions receive unprocessed arguments. Builtin functions may impose certain requirements on their arguments (like `USECOUNTER` requiring the name of a counter) and these requirements are checked on the arguments as received.

Summarizing: `EVAL` acts as follows:

- Its argument is presented to Yodl's lexical scanner
- The output produced by the processing of the argument is then inserted into the input stream *in lieu of* the original `EVAL` call.

Most built-in functions do *not* evaluate their arguments. In fact, only `ERROR, EVAL, FPUTS, INTERNALINDEX, PUSHSUBST, TYPEOUT, UPPERCASE, WARNING` and the iinternally used `XXSUBST` functions evaluate their arguments.

Postponing evaluations allows you to write:

```
DEFINESYMBOL(later)(SYMBOLVALUE(earlier))
```

Eventually, and not when `later` is defined, a statement like

```
SYMBOLVALUE(later)
```

produces the value of `earlier` at the moment `SYMBOLVALUE(later)` is processed. This is, in all its complex consequences, what would be expected in most cases. It allows us to write general macros producing output that is only evaluated when the text of symbols and values of arguments become eventually, rather than when the macro is defined, available.

Decisions like these invariably result in questions like 'what if I have to define variables using values of other variables?' In those cases `EVAL()` must be used. The following example shows the definition of three symbols: `one` receives an initial value, `two` returns `one`'s actual value when `two`'s value is displayed, `three`, using `EVAL()`, stores `one`'s initial value.

The example also shows yet another way to suppress macro calls, using the macro `nop()` which is defined in the all standard conversion types:

```
DEFINESYMBOL(one)(One's first value)
DEFINESYMBOL(two)(SYMBOLVALUE(one))
EVAL(DEFINESYMBOL+nop()(three)(SYMBOLVALUE(one)))
SETSYMBOL(one)(One's second value)
"SYMBOLVALUE(two)"      COMMENT(displays "One's second value")
"SYMBOLVALUE(three)"    COMMENT(displays "One's first value")
```

3.1.20 FILENAME

The function `FILENAME()` produces an absolute path to the currently processed Yodl file. This is not necessarily the *canonical* path name, as it may contain current- and parent-path directories.

3.1.21 FPUTS

The function `FPUTS` expects two arguments: the first argument is information to be appended to a file, whose name is given as the second argument. The first argument is processed by Yodl before it is appended to the requested filename, so it may contain macro calls.

For example, the following statement appends a countervalue to the mentioned file:

```
FPUTS(There have been COUNTERVALUE(attempts) attempts)(/tmp/logfile)
```

The second argument (name of the file) is not evaluated, but is used as received.

3.1.22 IFBUILTIN

The `IFBUILTIN` function tests whether its first argument is the name of a builtin function. If so, the second argument is evaluated, else, the third argument is evaluated. All three arguments (the variable, the true-list and the false-list) must be present; though the true-list and/or the false-list may be empty.

Example:

```
IFBUILTIN(IFBUILTIN)(\
    'BUILTIN' is a builtin - function
)(\
    'BUILTIN' is NOT a builtin - function
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.23 IFCHARTABLE

The `IFCHARTABLE` function tests whether its first argument is the name of a character table. The character table needs not be active. If the name is the name of a character table, the second argument is evaluated, else, the third argument is evaluated. All three arguments (the name, the true list and the false list) must be present; though the true list and/or the false list may be empty.

Example:

```
IFCHARTABLE(standard)(\
    'standard' is a character tablebuiltin - function
)(\
    'standard' is NOT a character tablebuiltin - function
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.24 IFDEF

The `IFDEF` function tests for the definition status of the argument in its first argument. If it is a defined entity, the second argument is evaluated, else, the third argument is evaluated. All three arguments (the entity, the true list and the false list) must be present; though the true list and/or the false list may be empty.

The true list is evaluated if the first argument is the name of:

- a built-in function, or
- a character table, or
- a counter, or
- a no-user-macro symbol, or
- a symbol, or
- a user-defined macro, or

Example:

```
IFDEF(someName)(\
    ‘someName’ is a defined entity
)(\
    ‘someName is not defined.
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.25 IFEMPTY

`IFEMPTY` expects three arguments: a symbol, a true-list and a false-list. `IFEMPTY` evaluates to the true-list if the symbol is an empty string; otherwise, it evaluates to the false-list.

The function does not further evaluate its argument. Its use is primarily to test whether a macro has received an argument or not. If the intent is to check whether a symbol's value is empty or not, IFSTREQUAL 3.1.30 should be used, where the first argument is the name of a symbol, and the second argument is empty.

Example:

```
IFEMPTY(something)(\
    ‘something’ is empty...
)(\
```

```
        'something' is not an empty string
    )
```

In the same way, `IFEMPTY` can be used to test whether an argument expands to a non-empty string. A more elaborate example follows below. Say you want to define a `bookref` macro to typeset information about an author, a book title and about the publisher. The publisher information may be absent, the macro then typesets `unknown`:

```
\
    DEFINEMACRO(bookref)(3)(\
        Author(s):      ARG1
        Title:          ARG2
        Published by:   \
        IFEMPTY(ARG3)
        (\
            Unknown\
        )(\
            ARG3\
        )
    )
```

Using the macro, as in:

```
\
    bookref(Helmut Leonhardt)
           (Histologie, Zytologie und Microanatomie des Menschen)
           ()
```

would now result in the text `Unknown` behind the `Published by:` line.

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.26 IFEQUAL

`IFEQUAL` expects four argument lists. It tests whether its first argument is equal to its second argument. If so, the third argument is evaluated, else, the fourth argument is evaluated. All four argument lists must be present, though all can be empty.

The first two arguments of `IFEQUAL` should be integral numeric arguments. In order to determine whether the first two arguments are equal, their values are determined:

- If the argument starts with an integral numerical value, that value is the value of the argument.

- If the argument is the name of a counter, the counter's value is the value of the argument
- If the values of the first two arguments van be determined accordingly, their equality determines whether the true list (when the values are equal) or the false list (when the values are unequal) will be evaluated.
- Otherwise, `IFEQUAL` evaluates the false list.

Example:

```
IFEQUAL(0)()(\
    0 and an empty string are equal
)(\
    0 and an empty string are not equal
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.27 IFGREATER

`IFGREATER` expects four argument lists. It tests whether its first argument is greater than its second argument. If so, the third parameter list is evaluated, otherwise its fourth argument is evaluated. All four argument lists must be present, though all can be empty.

The first two arguments of `IFGREATER` should be integral numeric arguments. In order to determine whether the first two arguments are equal, their values are determined:

- If the argument starts with an integral numerical value, that value is the value of the argument.
- If the argument is the name of a counter, the counter's value is the value of the argument
- If the values of the first two arguments van be determined accordingly, their order relation determines whether the true list (when the first value is greater than the second value) or the false list (when the first value is smaller or equal than the second value) is evaluated.
- Otherwise, `IFGREATER` evaluates the false list.

Example:

```
IFGREATER(counter)(5)(\
```

```
    counter exceeds the value 5
)(\
    counter does not exceeds the value 5, or counter is no Yodl-counter.
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.28 IFMACRO

The `IFMACRO` function tests whether its first argument is the name of a macro. If the name is the name of a macro, the second argument is evaluated, else, the third argument is evaluated. All three arguments (the name, the true list and the false list) must be present; though the true list and/or the false list may be empty.

Example:

```
IFMACRO(nested)(\
    ‘nested’ is the name of a macro
)(\
    There is no macro named ‘nested’
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.29 IFSMALLER

`IFSMALLER` expects four argument lists. It tests whether its first argument is smaller than its second argument. If so, the third parameter list is evaluated, otherwise its fourth argument is evaluated. All four argument lists must be present, though all can be empty.

The first two arguments of `IFSMALLER` should be integral numeric arguments. In order to determine whether the first two arguments are equal, their values are determined:

- If the argument starts with an integral numerical value, that value is the value of the argument.
- If the argument is the name of a counter, the counter's value is the value of the argument
- If the values of the first two arguments van be determined accordingly, their order relation determines whether the true list (when the first value is smaller

than the second value) or the false list (when the first value is greater than or equal to the second value) is evaluated.

- Otherwise, `IFSMALLER` evaluates the false list.

Example:

```
IFSMALLER(counter)(5)(\
    counter is smaller than the value 5, or counter is no Yodl-counter
)(\
    counter exceeds the value 5
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

3.1.30 IFSTREQUAL

`IFSTREQUAL` tests for the equality of two strings. It expects four arguments: two strings to match, a true list and a false list. The true list is only evaluated when the contents of the two string arguments exactly match.

The first two arguments of `IFSTREQUAL` are partially evaluated:

- If the argument is the name of a symbol, the symbol's value is the value of the argument
- Otherwise, the argument itself is used.

In the degenerate case where the string to be compared is actually the name of a `SYMBOL`, use a temporary `SYMBOL` variable containing the name of that symbol, and compare it to whatever you want to compare it with. Alternatively, write a blank space behind the arguments, since the arguments are then interpreted 'as is'. In practice, the need for these constructions seem to arise seldom, however.

Example:

```
IFSTREQUAL(MYSYMBOL)(Hello world)(
    The symbol 'MYSYMBOL' holds the value 'Hello world'
)(
    The symbol 'MYSYMBOL' doesn't hold the value 'Hello world'
)
```

3.1.31 IFSTRSUB

`IFSTRSUB` tests whether a string is a sub-string of another string. It acts similar to IFSTREQUAL, but it tests whether the second string is part of the first one.

The first two arguments of `IFSTREQULA` are partially evaluated:

- If the argument is the name of a symbol, the symbol's value is the value of the argument
- Otherwise, the argument itself is used.

In the degenerate case where the string to be compared is actually the name of a `SYMBOL`, use a temporary `SYMBOL` variable containing the name of that symbol, and compare it to whatever you want to compare it with. Alternatively, write a blank space behind the arguments, since the arguments are then interpreted 'as is'. In practice, the need for these constructions seem to arise seldom, however.

Example:

```
IFSTRSUB(haystack)(needle)(
    'needle' was found in 'haystack'
)(
    'needle' was not found in 'haystack'
)
```

Note that both 'haystack' and 'needle' may be the names of symbols. If they are, their contents are is compared, rather than the literal names 'haystack' and 'needle'

3.1.32 IFSYMBOL

The `IFSYMBOL` function tests whether its first argument is the name of a symbol. If it is the name of a symbol, the second argument is evaluated, otherwise the third argument is evaluated. All three arguments (the name, the true list and the false list) must be present; though the true list and/or the false list may be empty.

Example:

```
IFSYMBOL(nested)(\
    'nested' is the name of a symbol
)(\
    There is no symbol named 'nested'
)
```

Please note the preferred layout: The first argument immediately follows the function name, then the second argument (the *true list*) is indented, as is the *false list*. The layout closely follows the preferred layout of `if-else` statements of many programming languages.

C(FBB consider additive expressions)

3.1.33 IFZERO

`IFZERO` expects three arguments. If the first argument is zero (0) the function expands to the true list (the second argument). Otherwise it expands to the false list (the third argument).

The first argument of `IFZERO` should be an integral numeric value. Its value is determined as follows:

- If the argument starts with an integral numerical value, that value is the value of the argument.
- If the argument is the name of a counter, the counter's value is the value of the argument
- Otherwise, the first arguments evaluates as 0, and the false list is used.

Note that, starting with Yodl version 2.00 the first argument is not evaluated. So `COUNTERVALUE(somecounter)` always evaluates as 0. If the value of a counter is required, simply provide its name as the first argument of the `IFZERO` function.

Example:

```
DEFINEMACRO(environment)(2)(\
    IFZERO(ARG2)(\
        NOEXPAND(\end{ARG1})\
    )(\
        NOEXPAND(\begin{ARG1})\
    )\
)
```

Such a macro may be used as follows:

```
environment(center)(1)
    Now comes centered text.
environment(center)(0)
```

which would of course lead to `\begin` and `\end{center}`. The numeric second argument is used here as a on/off switch.

3.1.34 INCLUDEFILE

`INCLUDEFILE` takes one argument, a filename. The file is processed by Yodl. If a file should be inserted without processing the builtin function NOEXPANDINCLUDE 3.1.38 or NOEXPANDPATHINCLUDE 3.1.39 should be used.

The `yodl` program supplies, when necessary, an extension to the filename. The supplied extension is `.yo`, unless defined otherwise during the compilation of the program.

Furthermore, Yodl tries to locate the file in the Yodl's include path (which may be set using the `+NOTRANS(-{}-{})include` option). The actual value of the include path is shown in the usage information, displayed when Yodl is started without arguments.

Example:

```
INCLUDEFILE(latex)
```

Here, Yodl attempts to include the file `latex` or `latex.yo` from the current include path. When the file is not found, Yodl aborts.

3.1.35 INCWSLEVEL

`INCWSLEVEL` requires one (empty) argument.

It increments the current white-space level. The white-space level typically is used in files that only define Yodl macros. When no output should be generated while processing these files, the white-space level can be used to check for this. If the white-space level exceeds zero, a warning is generated if the file produces non-whitespace output. The builtin function `DECWSLEVEL` is used to decrement the whitespace level following a previous `INCWSLEVEL` call.

Once the white space level exceeds zero, no output is generated. White space, therefore is effectively ignored. The white space level cannot be reduced to negative values. A warning is issued if that would have happened if it were allowed.

Example:

```
INCWSLEVEL()
DEFINESYMBOL(....)
DEFINEMACRO(...)(...)(...)
DECWSLEVEL()
```

Without the `INCWSLEVEL` and `DECWSLEVEL`, calls, the above definition would generate four empty lines to the output stream.

The `INCWSLEVEL` and `DECWSLEVEL` calls may be nested. The best approach is to put an `INCWSLEVEL` at the first line of a macro-defining Yodl-file, and a matching `DECWSLEVEL` call at the very last line.

3.1.36 INTERNALINDEX

`INTERNALINDEX` expects one argument list. The argument list is evaluated and written to the index file.

The index file is defined since Yodl version 2.00, and contains the fixup information which was previously written to Yodl's output as the `.tt(Yodl)TAGSTART. ... .tt(Yodl)TAGEND.` sequence.

The index file allows for greater processing speed, at the expense of an additional file. The associated `yodlpost` postprocessing program reads and processes the index file, and modifies the corresponding yodl-output accordingly.

The index file is not created when output is written to the standard output name, since Yodl is unable to request the system for the current file offset.

The entries of the index file always fit on one line. `INTERNALINDEX` changes newline characters in its argument into single blank spaces. Each line starts with the current offset of Yodl's output file, thus indicating the exact location where a fixup is requested. An example of a produced fixup line could be

```
3004 ref MACROPACKAGE
```

indicating that at offset 3004 in the produced output file a reference to the label `MACROPACKAGE` is requested. Assuming a html conversion, The postprocessor thereupon writes something like

```
<a href="outfile04.html#MACROPACKAGE">4.3.2.</a>
```

into the actual output file while processing Yodl's output up to offset location 3004.

Consequently, producing Yodl-output normally consists of two steps:

- First, Yodl itself is started, producing, e.g., `out.idx` (the index file) and `out.yodl` (Yodl's raw output).
- Then, Yodl's post-processor processes `out.idx` and `out.yodl`, producing one or more final output files, in which the elements of the index file have been properly handled. This may result in multiple output file, like `report.html, report01.html, report02.html` etc.

3.1.37 NOEXPAND

`NOEXPAND` is used to send text to the final output file without being expanded by Yodl (the other methods are the `CHAR` macro, see section 3.1.4, and the `NOTRANS` macro, see section 3.1.40). `NOEXPAND` takes one argument, the text in question. Whatever occurs in the argument is not subject to parsing or expansion by Yodl, but is simply copied to the output file (except for `CHAR` and (iinternally used) `XXSUBST` functions in the argument, which *are* expanded. If `CHAR`-expansion is not required either NOTRANS 3.1.40 can be used).

Furthermore, the contents of the argument are also subject to character table translations, using the currently active table. This should come as no surprise. Ignoring

character tables would make both the processing of `CHAR` calls and the `NOTRANS` function superfluous.

So, the following situations are recognized:

	support chartables and CHAR	
Macro expansion	yes	no
Yes	(standard)	Push chartable (standard) Pop chartable
No	NOEXPAND	NOTRANS

E.g., let's assume that you need to write in your document the following text:

```
INCLUDEFILE(something or the other)
IFDEF(onething)(
    ...
)(
    ....
)
NOEXPAND(whatever)
```

The way to accomplish this is by prefixing the text by `NOEXPAND` followed by an open parenthesis, and by postfixing it by a closing parenthesis. Otherwise, the text would be expanded by Yodl while processing it (and would lead to syntax errors, since the text isn't correct in the sense of the Yodl language).

For this function, keep the following caveats in mind:

- There is only one thing that a `NOEXPAND` cannot protect from expansion: an `ARG`x in a macro definition. The argument specifier is always processed. E.g., after

  ```
  DEFINEMACRO(thatsit)(1)(
      That is --> NOEXPAND(ARG1) <-- it!
  )
  thatsit(after all)
  ```

  the `ARG1` inside the `NOEXPAND` statement is replaced with `after all`.

- The `NOEXPAND` function must, as all functions, be followed by a argument. The parentheses of the list must therefore be 'balanced'. For unbalanced lists, use `CHAR(40)` to set an open parenthesis, or `CHAR(41)` to typeset a closing parenthesis.

3.1.38 NOEXPANDINCLUDE

`NOEXPANDINCLUDE` takes one argument, a filename. The file is included.

The filename is uses as specified. The include path is not used when locating this file.

The argument to `NOEXPANDINCLUDE` is partially evaluated:

- If the argument is the name of a symbol, the symbol's value is the value of the argument
- Otherwise, the argument itself is used.

The thus obtained file name is not further evaluated: in particular, it is not affected by available character translations.

The contents of the file are included literally, not subject to macro expansion. Character translations are performed, though. If character translations are not appropriate, PUSHCHARTABLE can be used to suppress character table translations temporarily.

The purpose of NOEXPANDINCLUDE is to include source code literally in the document, as in:

```
NOEXPANDINCLUDE(literal.c)
```

The function NOEXPANDPATHINCLUDE can be used to insert a file which *is* located in one of the directories specified in Yodl's include path.

3.1.39 NOEXPANDPATHINCLUDE

`NOEXPANDPATHINCLUDE` takes one argument, a filename. The file is included. The file is searched for in the directories specified in Yodl's includepath.

The argument to `NOEXPANDPATHINCLUDE` is partially evaluated:

- If the argument is the name of a symbol, the symbol's value is the value of the argument
- Otherwise, the argument itself is used.

The thus obtained file name is not further evaluated: in particular, it is not affected by available character translations.

Like the `NOEXPANDINCLUDE` function, the contents of the file are included literally, not subject to macro expansion. Character translations are performed, though. If character translations are not appropriate, PUSHCHARTABLE 3.1.53 can be used to suppress character table translations temporarily.

The purpose of NOEXPANDPATHINCLUDE is to include source code as defined in a macro package literally into the document, as in:

```
NOEXPANDPATHINCLUDE(rug-menubegin.xml)
```

3.1.40 NOTRANS

NOTRANS copies its one argument literally to the output file, without expanding macros in it and without translating the characters with the current translation table. The NOTRANS function is typically used to send commands for the output format to the output file.

For example, consider the following code fragment:

```
COMMENT(--- Define character translations for \, { and } in LaTeX. ---)
DEFINECHARTABLE(standard)(
    '\\'    =    "$\\backslash$"
    '{'     =    "\\verb+{+"
    '}'     =    "\\verb+}+"
)

COMMENT(--- Activate the translation table. ---)
USECHARTABLE(standard)

COMMENT(--- Now two tests: ---)

NOEXPAND(\input{epsf.tex})
NOTRANS(\input{epsf.tex})
```

NOEXPAND sends

```
$\backslash$input\verb+{+epsf.tex\verb+}+
```

since the characters in its argument are translated with the standard translation table. In contrast, NOTRANS sends \input{epsf.tex}.

The argument of NOTRANS *must* be balanced with respect to its parentheses. When using an unbalanced set of parentheses, use CHAR(40) to send a literal (, or CHAR(41) to send a).

While converting Yodl-documents to target document types Yodl frequently uses the (not further documented) builtin function XXSUBST. In the unlikely event that the text XXSUBST(...) must be written in a document, the sequence

```
XXSUBST+CHAR(40)...CHAR(41)
```

can be used.

The NOEXPAND description summarizes all combinations of character translations and/or macro expansion, and how they are handled and realized by Yodl.

3.1.41 NOUSERMACRO

`NOUSERMACRO` controls `yodl`'s warnings in the following way: When Yodl is started with the `-w` flag on the command line, warnings are generated when Yodl encounters a possible macro name, i.e., a name that is followed by a parenthesized argument, while no macro by that name has been defined. Yodl then prints something like `cannot expand possible user macro`.

Examples of such sequences are, `The necessary file(s) are in /usr/local/lib/yodl`, or `see the manual page for sed(1)`. The candidate macros are `file` and `sed`; these names could just as well be 'valid' user macros followed by their argument.

When a corresponding `NOUSERMACRO` statement appears *before* `yodl` encounters the candidate macros, no warning is generated. A fragment might therefore be:

```
NOUSERMACRO(file sed)
The necessary file(s) are in ...
See the manual page for sed(1).
```

The `NOUSERMACRO` accepts one or more names in its argument, separated by white space, commas, colons, or semi-colons.

3.1.42 OUTBASE

`OUTBASE` inserts the current basename of the output file into the output file. The basename is the name of the file of which the directory components and extension were stripped.

If the output file is the standard output file, `-` is inserted.

3.1.43 OUTDIR

`OUTDIR` inserts the current path name of the output file into the output file. The path name is a, not necessarily absolute, designator of the directory in which the output file is located. If the output file is indicated as, e.g., `-o out`, then `OUTDIR` simply inserts a dot.

If the output file is the standard output file, a dot is inserted too.

3.1.44 OUTFILENAME

`OUTFILENAME` inserts the current filename of the output file into the output file. The filename is the name of the file of which the directory components were stripped.

If the output file is the standard output file, `-` is inserted.

3.1.45 PARAGRAPH

PARAGRAPH isn't really a builtin function, but as Yodl handles paragraphs in a special way it is probably useful to describe paragraph handling here nonetheless. Starting with Yodl 2.00 PARAGRAPH operates as follows:

If the macro is not defined, new paragraphs, defined as series of consecutive empty lines written to the output stream, are not handled different from any other series of characters sent to the output stream. I.e., they are inserted into that stream.

However, if the macro has been defined, Yodl calls it whenever a new paragraph (defined as a series of at least two blank lines) has been recognized.

The empty lines that were actually recognized may be obtained inside the PARAGRAPH macro from the XXparagraph symbol, *if* this symbol has been be defined by that time. If defined, it contains the white space that caused Yodl to call the PARAGRAPH macro.

Note that, in order to inspect XXparagraph it must have been defined first. Yodl itself does *not* define this symbol itself.

The PARAGRAPH macro should be defined as a macro not expecting arguments. The macro is thus given a chance to process the paragraph in a way that's fitting for the particular conversion type. If the PARAGRAPH macro produces series of empty lines itself, then those empty lines do *not* cause Yodl to activate PARAGRAPH. So, Yodl itself will not recursively call PARAGRAPH, although the macro could call itself recursively. Of course, such recursive activcation of PARAGRAPH is then the sole responsibility of the macro's author, and not Yodl's.

Some document languages do not need paragraph starts; e.g., LaTeX handles its own paragraphs. Other document languages do need it: typically, PARAGRAPH is then defined in a macro file to trigger some special action. E.g., a HTML converter might define a paragraph as:

```
DEFINEMACRO(PARAGRAPH)(0)(
    XXnl()
    NOTRANS(<p>)
)
```

A system like xml has more strict requirements. Paragraphs here must be opened and closed using pairs of and tags. In those cases an auxiliary counter can be used to indicate whether there is an open paragraph or not. The PARAGRAPH macro could check for this as follows, assuming the availability of a counter XXp:

```
DEFINEMACRO(PARAGRAPH)(0)(
    XXnl()
    IFZERO(XXp)(
    )(
        NOTRANS(</p>)
    )
    NOTRANS(<p>)
```

```
        SETCOUNTER(XXp)(1)
    )
```

Note that the above fragment exemplifies an approach, not necessarily *the* implementation of the `PARAGRAPH` macro for an xml-converter.

3.1.46 PIPETHROUGH

The builtin function `PIPETHROUGH` is, besides `SYSTEM`, the second function with which a Yodl document can affect its environment. `PIPETHROUGH` can be very useful. It uses an external program to accomplish special features. The idea is that an external command is started, to which a block of text from within a Yodl document is 'piped'. The output of that child program is piped back into the Yodl document; hence, a block of text is 'piped through' an external program. Whatever is received again in the Yodl run, is further processed.

The `PIPETHROUGH` function takes two arguments:

- the command to run, and
- the text to send to that command.

Functionally, the occurrence of the `PIPETHROUGH` function and of its two arguments is replaced by whatever the child program produces on its standard output.

An example might be the inclusion of the current date, as in:

```
The current date is:
PIPETHROUGH(date)()
```

In this example the command is `date` and the text to send to that program is empty.

The main purpose of this function is to provide a way by which external programs can be used to create, e.g., tables or figures for a given output format. Further releases of Yodl may contain such dedicated programs for the output formats.

3.1.47 POPCHARTABLE

Character tables which are pushed onto the table stack using `PUSHCHARTABLE()` are restored (popped) using `POPCHARTABLE()`. For a description of this mechanism please refer to section 2.3.3.

3.1.48 POPCOUNTER

`POPCOUNTER` is used to remove the topmost counter from the counter stack. The values of counters may be pushed on a stack using PUSHCOUNTER 3.1.54. To remove the topmost element of a counter's stack `POPCOUNTER` is available. `POPCOUNTER` expects one argument: the name of the counter to pop. The previously pushed value

then becomes the new value of the counter. A counter's value may be popped after defining it, whereafter the stack is empty, but the counter will still be defined. In that case, using the counter's value is considered an error.

Examples:

```
DEFINECOUNTER(YEAR)(1950)
POPCOUNTER(YEAR)
COMMENT(YEAR now has an undefined value)
```

See also section 2.5.

3.1.49 POPMACRO

`POPMACRO` is used to remove the actual macro definition, restoring a previously pushed definition. The values of macros may be pushed on a stack using PUSHMACRO.

To remove the topmost element of a macro's stack `POPMACRO` is available. `POPMACRO` expects one argument: the name of the macro to pop. The previously pushed value then becomes the new value of the macro.

A macro's value may be popped after defining it, after which its stack is empty. In that case, using the macro (although the macro's name is still defined) is considered an error.

Example:

```
DEFINEMACRO(Hello)(1)(Hello, ARG1, this is a macro definition)
Hello(Karel)
PUSHMACRO(Hello)(1)(Hello, ARG1, this is the new definition)
Hello(Karel)
POPMACRO(Hello)
Hello(Karel)
COMMENT(The third activation of Hello() produces the same output
        as the first activation)
```

3.1.50 POPSUBST

`POPSUBST` is used to revert to a previous level of interpretation of `SUBST` definitions. Refer to the descriptions of the `PUSHSUBST` and `SUBST` builtin commands below for details.

There is no limit to the number of times `POPSUBST` can be called. Once the '`PUSHSUBST` stack' is empty `SUBST` definitions are automatically interpreted (so no stack-underflow error is ever encountered).

3.1.51 POPSYMBOL

POPSYMBOL is used to remove the topmost symbol from the symbol stack. The values of symbols may be pushed on a stack using PUSHSYMBOL 3.1.57. To remove the topmost element of a symbol's stack POPSYMBOL is available.

POPSYMBOL expects one argument: the name of the symbol to pop. The previously pushed value then becomes the new value of the symbol.

A symbol's value may be popped after defining it, after which its stack is empty. In that case, using the symbol (although the symbol's name is still defined) is considered an error.

Example:

```
DEFINESYMBOL(YEAR)(This happened in 1950)
POPSYMBOL(YEAR)
COMMENT(YEAR now has an undefined value)
```

3.1.52 POPWSLEVEL

POPWSLEVEL is used to remove the topmost wslevel from the wslevel stack. The values of wslevels may be pushed on a stack using PUSHWSLEVEL 3.1.58. See also section DECWSLEVEL 3.1.8

To remove the topmost element of a wslevel's stack POPWSLEVEL is available. POPWSLEVEL expects one argument: the name of the wslevel to pop. The previously pushed value then becomes the new value of the wslevel. A wslevel's value may be popped after defining it, emptying the stack, but the wslevel will still be defined. In that case, using the wslevel's value is considered an error.

Example:

```
COMMENT(Assume WS level is zero)

PUSHWSLEVEL(1)
COMMENT(WS level now equals 1)

POPWSLEVEL()
COMMENT(WS level now equals 0 again)
```

3.1.53 PUSHCHARTABLE

Once a character table has been defined, it can be *pushed* onto a stack using PUSHCHARTABLE. The pushed chartable may be *popped* later. PUSHCHARTABLE is described in more detail in section 2.3.3.

3.1.54 PUSHCOUNTER

PUSHCOUNTER is used to start another lifetime for a counter, pushing its current value on a stack. A stack is available for each individual counter.

PUSHCOUNTER expects two arguments: the name of the counter to push and an additive expression whose value becomes the counter's new value (after pushing the current value)

The additive expression may not contain blank spaces and may use + and - operators, its operands may either be integral numeric values or names of (defined) counters. The resulting value can be negative; in that case, a value is subtracted from the destination counter.

When the second argument is empty, then the new value will be zero. Specify the name of the counter twice to merely push its value, without modifying its current value.

Examples:

```
DEFINECOUNTER(YEAR)(1950)
PUSHCOUNTER(YEAR)(1962)
COMMENT(YEAR now has the value 1962, and a pushed value of 1950)
```

See also section 2.5.

3.1.55 PUSHMACRO

PUSHMACRO is used to start another lifetime for a macro, pushing its current definition on a stack. A stack is available for each individual macro.

PUSHMACRO expects three arguments: the name of the macro to push, the number of its arguments after pushing (which may be different from the number of arguments interpreted by the pushed macro) and its new definition.

So, PUSHMACRO is used exactly like DEFINEMACRO, but redefines a current macro (or define a new macro if no macro was defined by the name specified as its first argument.

Example:

```
DEFINEMACRO(Hello)(1)(Hello, ARG1, this is a macro definition)
Hello(Karel)
PUSHMACRO(Hello)(1)(Hello, ARG1, this is the new definition)
Hello(Karel)
POPMACRO(Hello)
Hello(Karel)
COMMENT(The third activation of Hello() produces the same output
        as the first activation)
```

3.1.56 PUSHSUBST

PUSHSUBST can be used to (temporarily) suppress the interpretation of SUBST definitions (the SUBST built-in command is covered below, refer to its description for an example).

PUSHSUBST expects one argument: an integral number which is either 0 or non-zero (commonly: 1). After calling PUSHSUBST(0) SUBST definitions are not interpreted anymore; use POPSUBST() to revert to the previous type of interpretation. Alternatively, PUSHSUBST(0) can be used to stack another level of SUBST interpretations on top of the last-used one.

On a 64-bit computer the PUSHSUBST stack can hold slightly more than 60 SUBST interpretation levels. When more levels are pushed, the oldest levels are silently forgotten. Calling POPSUBST once the PUSHSUBST stack is empty results in activating the SUBST interpretations (and so a stack-underflow error will not be encountered).

3.1.57 PUSHSYMBOL

PUSHSYMBOL is used to start another lifetime for a symbol, pushing its current value on a stack. A stack is available for each individual symbol.

PUSHSYMBOL expects two arguments: the name of the symbol to push and its new text after pushing. When the second argument is an empty argument, the new text will be empty. The new text may be specified as a literal text, or as the name of an existing symbol. Specify the name of the symbol twice to merely push its value, without modifying its current value.

Examples:

```
DEFINESYMBOL(YEAR)(This happened in 1950)
PUSHSYMBOL(YEAR)(This happened in 1962)
COMMENT(YEAR now has the value ‘This happened in 1962’ and a
        pushed value of ‘This happened in 1950’)
```

3.1.58 PUSHWSLEVEL

PUSHWSLEVEL is used to start another lifetime of the white-space level pushing the level's current value on a stack. See also section INCWSLEVEL 3.1.35

PUSHWSLEVEL expects one argument, the new value of the white-space level. This value may be specified as a numerical value or as the name of a counter. The argument may be empty, in which case the new value will be zero.

Example:

```
COMMENT(Assume WS level is zero)

PUSHWSLEVEL(1)
```

```
COMMENT(WS level now equals 1)

POPWSLEVEL()
COMMENT(WS level now equals 0 again)
```

3.1.59 RENAMEMACRO

`RENAMEMACRO` takes two arguments: the name of a built-in macro (such as `INCLUDEFILE`) and its new name.

E.g., after

```
RENAMEMACRO(INCLUDEFILE)(include)
```

a file *must* be included by `include(file)`. `INCLUDEFILE` can no longer be used for this: following the `RENAMEMACRO` action, the old name can no longer be used; it becomes an undefined symbol.

If you want to make an *alias* for a built-in command, do it with `DEFINEMACRO`. E.g., after:

```
DEFINEMACRO(include)(1)(INCLUDEFILE(ARG1))
```

both `INCLUDEFILE` and `include` can be used to include a file.

3.1.60 SETCOUNTER

`SETCOUNTER` expects two arguments: the name of a counter, and an additive expression defining the value to assign. The counter must be previously created with `DEFINECOUNTER`.

The additive expression may not contain blank spaces and may use + and - operators, its operands may either be integral numeric values or names of (defined) counters. The resulting value can be negative; in that case, a negative value is assigned to the destination counter.

For example, if `one` and `two` are counters, then

```
SETTOCOUNTER(one)(-two)\//    assigns -two's value to one
SETTOCOUNTER(one)(two+two)\// assigns 2 x two's value to one
```

See also section 2.5.

3.1.61 SETSYMBOL

SETSYMBOL expects two arguments: the name of a symbol, and the text to assign to the named symbol. The symbol must previously have been defined by DEFINESYMBOL.

3.1.62 SUBST

SUBST is a general-purpose substitution mechanism for strings appearing in the input. SUBST takes two arguments: a search string and a substitution string. E.g., after

```
SUBST(VERSION)(1.00)
```

Yodl transforms all occurrences of VERSION in its input into 1.00.

SUBST is also useful in situations where multi-character sequences should be converted to accented characters. E.g., a LaTeX converter might define:

```
SUBST('e)(+NOTRANS(\'{e}))
```

Each 'e in the input will subsequently be converted to +NOTRANS(\'{e}).

SUBST may be used in combination with the command line flag -P, as in a invocation

```
yodl2html -P'SUBST(VERSION)(1.00)' myfile.yo
```

Another useful substitution might be:

```
SUBST(_OP_)(CHAR(40))
SUBST(_CP_)(CHAR(41))
```

which defines an opening parenthesis (_OP_) and a closing parenthesis (_CP_) as mapped to the CHAR function. The strings _OP_ and _CP_ might then be used to produce unbalanced arguments.

Note that:

- The first argument of the SUBST command, the search string, is taken literally. Yodl does not expand it; the string must be literally matched in the input.
- The second argument, the replacement, is further processed by Yodl. Protect this text by NOTRANS or NOEXPAND where appropriate.

Substitutions occur extremely early while `Yodl` processes its input files. In order to process its input files, `Yodl` takes the following steps:

1. It requests input from its lexical scanner (so-called *tokens*)
2. Its parser processes the tokens produced by the lexical scanner
3. Its parser may send text to an output 'object', which eventually appears in the output file generated by `Yodl`.

`Yodl` performs all macro substitutions in step 2, and all character table conversions in step 3. However, the lexical scanner has access to the `SUBST` definitions: as soon as its lexical analyzer detects a series of characters matching the defining sequence of a `SUBST` definition, it replaces that defining sequence by its definition. That definition is then again read by the lexical scanner. Of course, this definition may, in turn, contain defining sequences of other `SUBST` definitions: these are then replaced by their definitions as well. This implies:

- Circular definitions may cause the lexical scanner to get stuck in a replacement loop. It is the responsibility of the author defining `SUBST` definitions to make sure that this doesn't happen.
- Neither the parser, nor the output object ever sees the `SUBST` defining character sequences: they only see their definitions.

In some cases substitutions must be suppressed. Consider double quoted text strings that are frequently used in programming languages. E.g., `"hello world"`. The text inside the string should not be converted by Yodl, but unless substitutions can be suppressed the string

```
"\"evil code"
```

appears as

```
"+NOTRANS(\"{e})vil code"
```

To suppress the interpretation of `SUBST` definitions `PUSHSUBST`, introduced earlier, can be used. The predefined macro `verb` suppresses the interpretation of `SUBST` definitions by starting with `PUSHSUBST(0)` and ending with `POPSUBST()`.

3.1.63 SYMBOLVALUE

`SYMBOLVALUE` expands to the value of a symbol. Its single argument must be the name of a symbol. The symbol must have been created earlier using `DEFINESYMBOL`. Example:

```
The symbol has value SYMBOLVALUE(MYSYMBOL).
```

3.1.64 SYSTEM

SYSTEM takes one argument: a command to execute. The command is run via the standard C function system.

SYSTEM can be useful in many ways. E.g., you might want to log when someone processes your document, as in:

```
SYSTEM(echo Document processed! | mail myself@my.host)
```

Note that SYSTEM merely performs an system-related task. It's a process that is separated from the Yodl process itself. One of the consequences of this is that any output generated by SYSTEM not normally appears into Yodl's output file. If the output of a subprocess should be inserted into Yodl's output file, either use PIPETHROUGH 3.1.46, or insert a temporary file as shown in the following example:

```
SYSTEM(date > datefile)
The current date is:
INCLUDEFILE(datefile)
SYSTEM(rm datefile)
```

3.1.65 TYPEOUT

TYPEOUT requires one argument. The text of the list is sent to the standard error stream, followed by a newline. This feature can be handy to show, e.g., messages such as version numbers in macro package files.

Example: The following macro includes a file and writes to the screen that this file is currently processed.

```
DEFINEMACRO(includefile)(1)(
    TYPEOUT(About to process document: ARG1)
    INCLUDEFILE(ARG1)
)
```

3.1.66 UPPERCASE

UPPERCASE converts a string or a part of it to upper case. It has two arguments:

- The string to convert;
- A length, indicating how many characters (starting from the beginning of the string) should be converted.

The length indicator can be smaller than one or larger than the length of the string; in that case, the whole string is convertered.

Example:

```
UPPERCASE(hello world)(1)
UPPERCASE(hello world)(5)
UPPERCASE(hello world)(0)
```

This code sample expands to:

```
Hello world
HELLO world
HELLO WORLD
```

3.1.67 USECHARTABLE

`USECHARTABLE` takes one argument: the name of a translation table to activate. The table must previously have been defined using `DEFINECHARTABLE`. See section 2.3 for a description of character translation tables.

Alternatively, the name may be empty in which case the default character mapping is restored.

3.1.68 USECOUNTER

`USECOUNTER` is a combination of `ADDTOCOUNTER` and `COUNTERVALUE`. It expects one argument: the name of an defined counter (see DEFINECOUNTER 3.1.10).

The counter is first incremented by 1. Then the function expands to the counter's value.

See also section 2.5.

3.1.69 VERBOSITY

`VERBOSITY` expects two arguments, and may be used to change the verbosity level inside `Yodl` files. The function may be used profitably for debugging purposes, to debug the expansion of a macro or the processing of a `Yodl` input file.

The first argument indicates the processing mode of the second argument, and it may be:

- Empty, in which case the message-level is set to the value specified in the second argument;

- `+`, in which case the value specified in the second argument augments the current message level;
- `-`, in which case the value specified in the second argument augments is removed from the current message level

The second argument specifies one or more, separated by blanks, message level names or it may be set to a hexadecimal value (starting with `0x`), using hexadecimal values to represent message levels. Also, `NONE` may be used, to specify no message level, or `ALL` can be used to specify all message levels.

The following message levels are defined:

- ALERT (0x40). When an alert-error occurs, Yodl terminates. Here Yodl requests something of the system (like a `get_cwd()`), but the system fails.
- CRITICAL (0x20). When a critical error occurs, Yodl terminates. The message itself can be suppressed, but exiting can't. A critical condition is, e.g., the omission of an open parenthesis at a location where a parenthesized argument should appear, or a non-existing file in an `INCLUDEFILE` specification (as this file should be parsed). A non-existing file with a `NOEXPANDINCLUDE` specification is a plain (non-critical) error.
- DEBUG (0x01). Probably too much info, like getting information about each character that was read by Yodl.
- ERROR (0x10). An error (like doubly defined symbols). Error messages will not stop the parsing of the input (up to a maximum number of errors), but no output is generated.
- INFO (0x02). Not as detailed as 'debug', but still very much info, like information about media switches.
- NOTICE (0x04). Information about, e.g., calls to the builtin function calls.
- WARNING (0x08). Something you should know about, but probably not affecting Yodl's proper functioning

There also exists a level EMERG (0x80) which cannot be suppressed.

The value `0x00` represents `NONE`, the value `0xff` represents `ALL`.

When specifying multiple message levels using the hexadecimal form, their hexadecimal values should be binary-or-ed: adding them is ok, as long as you don't specify `ALL`:

```
VERBOSITY()(0x06)
COMMENT(this specifies 'INFO' and 'NOTICE')
```

When specifying message levels by their names, the names may be truncated at a unique point. However, the message level names are interpreted case sensitively, so `INF` for `INFO` is recognized as such, but `info` for `INFO` isn't. The following examples all specify verbosity levels INFO and NOTICE:

```
VERBOSITY()(I N)
VERBOSITY()(N I)
VERBOSITY()(NOT IN)
VERBOSITY()(INFO NOTICE)
```

3.1.70 WARNING

`WARNING` takes one argument: text to display as a warning. The `yodl` program makes sure that before showing the text, the current file and line number are printed. Other than this, `WARNING` works just as `TYPEOUT` (see section 3.1.65).

Note that an analogous function `ERROR` exists, which prints a message and then terminates the program (see section 3.1.18).

Chapter 4

Macros and Document types

The macro package distributed with `Yodl` is described in this chapter. The macro package consists of a number of definition files, which convert a Yodl document that follows a certain syntax to an output format. The main output formats, currently supported, are:

- HTML;
- LaTeX (plain LaTeX, no `latex2e`);
- The `groff` 'man' format which is used for man pages;
- The `groff` 'ms' format which is more expressive;
- Basic, plain text

The following conversion format is in an experimental stage:

- XML, as used by the University of Groningen's so-called 'webplatform'.

Currently discontinued conversion formats are:

- SGML, although the basic macros are available. SGML can probably be reactivated fairly quickly. Contact the maintainer if support for SGML should be reinstated
- texinfo, mainly due to the fact that the current maintainer doesn't know what the required *post-processing* actions are.
- tely, since this conversion format is unknown to the current maintainer.

Other formats may be available, but maybe in an unstable state. Contact the the maintainer if you have a new format to add, or want to reanimate formats that were previously available.

4.1 General structure of a Yodl document

This section describes the general format of a Yodl document.

First of all, a Yodl document needs a *preamble*. This part of the document must be at the top, and must define the modifiers and the document type. Modifiers, when present, must appear first.

Modifiers are often specific for a particular target document type (e.g., `latexoptions` or `mailto`), but may also have a general nature (e.g., `affiliation` or `abstract`). All modifiers are used to modify parameters of document types. Therefore, they must be specified before the document type is defined.

All modifiers are listed in section 4.3.7. In general, you should use as many modifiers as appropriate. E.g., you should define a `mailto` even when you're not planning to convert your document to HTML. The reason is twofold: first, you might later decide that a HTML version isn't a bad idea after all. Second, later versions of the converters might use `mailto` even for non-HTML output formats.

Following the modifiers, the *document type* is defined. The document type is either `article`, `report`, `book`, `plainhtml` or `manpage`. Except for the `manpage` document type, which is a highly specialized document type, described in section 4.1.2, the following rules apply:

A decision about the document type to use should be based on its complexity. If the document's organization becomes too complex, it is probably a good idea to use a document type supporting a more complex organization. E.g., a complex *article* might be written as an accessible *report*, combining related sections into chapters. Similarly, the structure of a report having 30 chapters might improve when it's reorganized as a *book* having parts. To offer a rule of thumb: a document should have no more than approximately ten top-level sections, and each top-level sectioning should have no more than approximately ten subsections, etc..

The document type influences the way Yodl formats the output. An `article` (or `plainhtml`) results in one output file. E.g., one final document when converting to HTML. If your article is way too long, then the loading of the HTML document also requires much time. When converting to HTML, Yodl splits `report`s and `book`s into files each holding a chapter. These can be accessed through the table of contents. So, the document length can also be relevant when you contemplate switching to a `report` or `book`.

Documents using special macros, must have defined these macros *before* they are used. An appropriate location for these macros is immediately beyond the preamble. E.g., see the file `Documentation/manual/manual.yo` distributed with the Yodl package. This is the main file of this manual, showing the preferred organization of Yodl files.

To answer *yes-but-what-if* oriented minds, here are two results of the wrong order of text, preamble and modifiers:

- If you put text before the preamble, i.e., before stating the document type, chances are that Yodl happily translates the file, but subsequent states probably fail. E.g., the <`html`> tag would come too late in a HTML conversion, causing the HTML browser to become confused. Or, the `\documentstyle` definition would be seen too late by the LaTeX typesetter.

- If you put modifiers, such as `latexoptions`, *beyond* the document type, then these modifiers have no effect; though Yodl won't complain either. The reason for this being that the definitions of such modifiers are encountered only **after** the stage where they were needed....

4.1.1 Document types

As distributed, Yodl supports four document types: *article, report, book* and the *manual* page. Note that document types are essentially unrelated to output formats; a book can be converted to each of the output formats, and a manual page can be converted to a `.html` file. Nevertheless, some formats are particularly useful for some document types. A book converted to the `man` output format to be processed later with `groff` won't look too good. Its looks would greatly improve when the document would be converted to ASCII using the `ms` output format.

Following the preamble and the definition of specialized macros symbols and counters, documents start by specifying the document type. The available macros are:

- `article(title)(author)(date)`: The `article` document type should be used for short documents. Its arguments specify the document's title, author and date.

  In articles, the title page is numbered and the table of contents is on the title page. The sectioning commands `sect`, `subsect` etc. are available.

- `report(title)(author)(date)`: The `report` document type differs from an `article` in that it has a separate unnumbered title page, a table of contents on a page of its own, and it supports the sectioning command `chapter` in addition to the ones supported by `article`s. A `report` should be used fir larger documents.

- `book(title)(author)(date)`: The `book` type is for even larger documents. In addition to the sectioning commands supported by `report` it supports the sectioning command `part`.

- `plainhtml(title)`: This document type is typically used in HTML output. It's implemented for situations where you only need to create a HTML file, but want to use Yodl to help you by providing useful macros. This document type is similar to `article`, but does not require you to specify `author` and `date` arguments (In fact, you can emulate `plainhtml` by using an `article`, using empty author and date arguments).

- `manpage(title)(section)(date)(source)(manual)`: The `manpage` document type should only be used to write Unix-style manual pages. It uses its own sectioning commands to reflect the necessary sections in a manual page. This document format is described separately in 4.1.2.

These macros provide, globally, three functions: First, the macros generate any commands that need to appear before 'real' text is sent to the output file. E.g., the LaTeX output needs a `\documentstyle` preamble, HTML output needs <`html`> and <`body`> tags.

Second, the macros define appropriate document-dependent settings. E.g., the LaTeX converter defines the title, author and date using `\title` etc..

Third, the actual document is started. E.g., for LaTeX this means a `\begin{`*type*`}`, followed by the appropriate commands to generate a the document title and the table of contents. The `title` setting in the above macros defines the document title which always appears on the front page of the document. For HTML output, this is also the title of the HTML file (or files), as appearing in the HTML <title> tag.

The fact that the macros defining the document type perform many functions means that once the macro is started, nothing 'extra' can be inserted between, e.g., the generated title and the table of contents. Sometimes this is not what you'd like; as is the case with an abstract. Yodl therefore uses *modifiers*, appearing *before* the document type macros, to insert information between the various elements of a document definition.

4.1.2 The manpage document type

The *manpage* document type was implemented to simplify the construction of Unix-style manual pages. `Yodl` mapage documents can be converted to `groff` documents (using `yodl2man`), to `html` documents (using `yodl2html`), or to plain ascii text documents (using `yodl2txt`).

A *manpage* document **must** be organized as follows:

1. The manual page itself is defined, using the macro

   ```
   manpage(short title)
               (section)
               (date)
               (source)
               (manual)
   ```

   Its arguments are:

   Short title: This should be the program name or something similar; i.e., whatever the manpage is describing.

   Section: A number, stating the manpage section. The Linux man (7) page recognizes the following manpage sections:

   - Section 1 is for commands, like `ls`;
   - Section 2 is for system calls, like `fork()`;
   - Section 3 is for library calls, like `strdup()`;
   - Section 4 is for special files (like *devices*);
   - Section 5 is for file formats, (like `syslog.conf`);
   - Section 6 is for games;
   - Section 7 is for macro packages and conventions;
   - Section 8 is for system management commands;
   - Section 9 is for other types of manpages, such as kernel commands.

   Date: The date of release.

   Source: The package where the manpage belongs to.

   Manual: The manual to which the package belongs.

The arguments of the *manpage* macro define, e.g., the headers and footers of the manual page. The `date`, `source` and `manual` arguments can be empty.

2. The subject of the manpage is defined using

   ```
   manpagename(name)(short description)
   ```

   The `name` argument should be a short name (e.g., the program name), and the `short description` should state the function. The descriptive argument is used by, e.g., the `whatis` database.

3. The synopsis starts after:

   ```
   manpagesynopsis()
   ```

   Following this, an abbreviated usage information is presented. This information should show, e.g., the possible program flags and required arguments; but no more.

4. The description is given after:

   ```
   manpagedescription()
   ```

   This is followed by some descriptive text. The descriptive text can e.g. show what the program (function, file, game, etc.) is supposed to do.

5. Options are expected after:

   ```
   manpageoptions()
   ```

   The options are typically a descriptive list of possible flags and their meaning. This section lists the information of the synopsis, but also gives an in-depth description. The `manpageoptions()` section is optional.

6. Necessary files are listed after:

   ```
   manpagefiles()
   ```

7. The 'see also' entry is defined by:

   ```
   manpageseealso()
   ```

   This is then followed by a list of related manual pages. Here, use the format `bf(topic)(sectionnr)`, e.g., `Yodl`(1).

8. Diagnostics are described after:

```
manpagediagnostics()
```

   Diagnostics can state, e.g., what error messages are produced by the program and what the cure is.

9. Known bugs should be mentioned after:

```
manpagebugs()
```

   This section is optional.

10. Finally, the author is stated after:

```
manpageauthor()
```

The `manpage` document type **requires** you to follow the above order of commands strictly and to state all the necessary sections (and optionally, to state the not required sections but in their proper sequence). Furthermore, sectioning commands that are available in other document types (`sect`, `subsect` etc.) are not allowed in a `manpage`. You *can* however insert other sections in the manual page with the macro `manpagesection`. This macro takes one argument: the title of the extra section. It is suggested that you type the section name in upper case, to conform to the standard.

As an example, the manual page for the `yodl` program follows (the actual manual page may differ):

```
manpage(yodl)
       (1)
       (1996)
       (The Yodl Package)
       (Yet oneOther Document Language)

manpagename(yodl)(main Yodl converter)

manpagesynopsis()
YODL [-DNAME] [-IDIR] [-oFILE] [-PCMD] [-pPASS] [-t] [-v] [-w] [-h]
[-?]  inputfile [inputfile...]

manpagedescription()
This manual page describes the YODL program, the main converter of the
Yodl package. This program is used by the bf(yodl2....) shell scripts,
e.g., bf(yodl2tex) or bf(yodl2html).

manpageoptions()
```

```
    description(
    dit(-DNAME) Defines symbol em(NAME).
    dit(-IDIR) Overrules the standard include directory (default
em(/usr/local/lib/yodl)) with em(DIR).
    dit(-oFILE) Specifies em(FILE) as the output file (default is stdout).
    dit(-PCMD) 'Preloads' command em(CMD), as if em(CMD) was the first line
of the input.
    dit(-pPASS) Defines em(PASS) as the maximum number of 'passes'; when this
number is exceeded, YODL aborts.
    dit(-t) Enables tracing mode. Useful for debugging.
    dit(-v) Raises the verbosity mode. Useful for debugging.
    dit(-w) Enables warning. When enabled, YODL warns when it encounters
inconsistencies.
    dit(-h, -?) Shows usage information.
    dit(inputfile) File to process, use em(-) to instruct YODL to read
from stdin.
    )

    manpagefiles()
    The YODL program requires no files, but 'normal' usage of the Yodl package
requires macro files installed (usually in bf(/usr/local/share/yodl)). The
files in this directory are included by the converters bf(yodl2txt) etc..

    manpageseealso()
    bf(yodl2tex), bf(yodl2html), bf(yodl2man), etc..

    manpagediagnostics()
    Warnings and errors of YODL are too many to enumerate, but all errors
are printed to em(stderr) after which YODL exits with a non-zero
status.

    manpagebugs()
    There may be bugs in the YODL program, but that's not very likely.
More likely you'll encounter bugs or omissions in the macro package
itself.

    manpageauthor()
    Karel Kubat
```

4.2 Predefined macros

This section describes all macros defined by default. Altering or removing these macros may produce unexpected results when converting `Yodl` documents to other formats. Furthermore, these macros often depend on macros or other symbols defined for internal use.

Many predefined macros depend on symbols start with `XX`. Therefore, it is strongly advised not to start any locally defined symbol with `XX` as doing so, or undefining existing symbols starting with `XX`, may also produce unexpected results.

Here are the default macros, alphabetically ordered:

4.2.1 abstract(text)

Defines an abstract for an article or report type of document. Abstracts are not implemented for books or manpages. Must appear **before** starting the document using the `article` or `report` macro.

4.2.2 addntosymbol(symbol)(n)(text)

Adds `text` `n` times to `symbol`. The value `n` may also be the name of a defined counter (which is not modified).

4.2.3 affiliation(site)

Defines an affiliation, to appear in the document titlepage below the author field. Must appear **before** starting the document with `article`, `report` or `book`. The affiliation is only printed when the author field is not empty. When converting to html the way the affiliation is displayed can be tuned using CSS id selector specifications. The affiliation has `id="affiliation"`.

4.2.4 AfourEnlarged()

Enlarges the usable height of A4 paper by 2 cm.: the top margin is reduced by 2 cm. This macro should be called in the preamble. The macro is available only for LaTeX conversions.
This macros is somewhat obsolete. Instead, using `PageEnlarged` is advised

4.2.5 appendix()

Starts appendices

4.2.6 article(title)(author)(date)

Starts an article. The top-level sectioning command is `(n)sect`. In HTML conversions only one output file is written, while the way the headings are displayed can be tuned using CSS id selector specifications: the title has `id="title"`, the author `id="author"`, and the date `id="date"`.)

4.2.7 attrib(text)

In html, pushes `text` as an attribute for the next html tag supporting `attrib`. E.g, to set a blue color and 30 pixel left-hand side margin for a section use

```
attrib(style="color:blue;margin-left:30px;")\
sect(Section name)
```

This results in the html markup

```
<h1 style="color:blue;margin-left:30px;">Section name</h1>
```

This macro is only effective with html conversions. It is applied in a stack-wise fashion: when multiple `attrib` calls are used, then the topmost attrib-string is added to the first macro calling the `attribinsert` macro, with subsequent macros using subsequent elements on the attrib-stack.

Commonly used attributes are `id="idname"`, expecting a `#idname` CSS label in either internal or external CSS specifications, or `style="spec"` (as shown in the example).

Example: when using

```
attrib(width = "100" height = "100")
attrib(id = "#fig")
figure(imgfile)(Caption)(IMG)
```

then the `#id` attribute is applied to <`figure`>, and the `width` and `height` attributes are applied to <`img`>, which html markup is inserted by the `figure` macro.

The `attrib` macro is supported by the following predefined macros (between parentheses the number of attribute strings that are inserted by these macros; if only 1 attribute string is inserted no number is shown):

`bf cell cells center chapter code dashes dit em figure(3) file htmltag itdesc lchapter link lref lsect lsubsect lsubsubsect nchapter npart nsect nsubsect nsubsubsect paragraph part quote row sc sect strong subs subsect subsubsect subsubsubsect sups tableatt tbl tac tc tnac tnc tr tt ttbegin url verb verborg verbinclude`.

4.2.8 attribclear()

Removes any existing contents from the attrib-stack. This macro is only active when converting to html

4.2.9 attribinsert()

In html, if the attrib-stack is not empty, inserts the value on top of the attrib-stack and then pops the topmost value. If the attrib-stack is empty, nothing happens.

4.2.10 bf(text)

Sets `text` in boldface.

4.2.11 bind(text)

Generate a binding character (non-breaking space) after text.

4.2.12 book(title)(author)(date)

Starts a book document. The top-level sectioning command is `(n)chapter`, `(n)part` being optional. In HTML output files are created for each chapter, while the way the headings are displayed can be tuned using CSS id selector specifications: the title has `id="title"`, the author `id="author"`, and the date `id="date"`.)

4.2.13 cell(contents)

Sets a table cell, i.e., one element in a row. With the man/ms converters multiple blanks between `cell()` macro calls are merged into a single blank character.

Instead of using `cell` in `table`, consider using `tc` in `tbl`.

4.2.14 cells(nColumns)(contents)

Set a table cell over `nColumns` columns. With LaTeX and xml the information in the combined cells is centered.

With man/ms conversions the `cells()` macro simply calls the `cell()` macro, but here the `setmanalign()` macro can be used to determine the alignment of multiple cells.

With html the macro `attrib` can be used, but when it contains a `style` specification the macro's default `style="text-align: center"` is ignored (but it can optionally be specified using the `attrib` macro).

Instead of using `cells` in `table`, consider using `tnc` in `tbl`.

4.2.15 cellsline(from)(count)

Sets a horizontal line starting at column number `from` over `count` columns in a row. If `from` is less then the number of columns already added to a row then it is ignored. This macro must be embedded in a `row` macro defining a table row. To put a line across the table's full width use `rowline`. To set horizontal lines across columns 1 until 2 and columns 4 until 5 table of a table use:

```
row(cellsline(1)(2)cellsline(4)(2))
```

Combining `cellsline` and `cell` or `cells` calls in one row produces undefined results.

Instead of using `cellsline` in `table`, consider using `tline` in `tbl`.

4.2.16 center(text)

Centers `text`. Use `nl()` in the text to break lines. In html the `attrib` macro is not supported, but a division (`div`) with style definition `text-align: center` is used. To center a table in html use the `tableatt` macro. If a `table` or `tableatt` macro is used inside a `center` macro then the contents of columns are column-wise centered.

Inside a `center(...)` context the counter `XXcenter` is unequal 0.

4.2.17 chapter(title)

Starts a new chapter in books or reports.

4.2.18 cindex()

Generate an index entry for LaTex() or texinfo c-indices. Its argument is the index entry. See also the `[fptv]index` macro.

4.2.19 cite(text)

Sets `text` as a citation or quotation

4.2.20 clearpage()

Starts a new page, when the output format permits. Under HTML a horizontal line is drawn.

4.2.21 code(text)

Sets `text` in code font, and prevents it from being expanded. For unbalanced parameter lists, use `CHAR(40)` to get `(` and `CHAR(41)` to get `)`.

4.2.22 columnline(from)(through)

Sets a horizontal line over some columns in a row. Note that `columnline` defines a row by itself, consisting of just a horizontal line spanning some of its columns, rather than the table's full width, like `rowline`. The two arguments represent column numbers. It is the responsibility of the author to make sure that the `from` and `through` values are sensible. I.e.,

```
1 <= from <= through <= ncolumns
```

To set a horizontal line in just one column select `through` equal to `from`.

Note: this macro cannot be used if multiple lines must be set in one row. In those cases the macros `tline, tskip`, and `tendline` should be used.

Instead of using `columnline` in `table`, consider using `tline` in `tbl`.

4.2.23 dashes()

Inserts two dashes in teletype font, and prevents them from being expanded.

In html the `attrib` macro is recognized by the <`code`> tag that is used to embed the two dashes.

4.2.24 def(macroname)(nrofargs)(redefinition)

Defines `macroname` as a macro, having `nrofargs` arguments, and expanding to `redefinition`. This macro is a shorthand for `DEFINEMACRO`. An error occurs when the macro is already defined. Use `redef()` to unconditionally define or redefine a macro.

4.2.25 description(list)

Sets `list` as a description list. Use `dit(item)` to indicate items in the list.

4.2.26 dit(itemname)

Starts an item named `itemname` in a description list. The list should be used in `description` macros. With `html` conversions the contents of a description item is separated from the item itself. The `dit` macro only defines the item, and not the description itself. This macro sets the item in bold-face ('strong' font). The macro `itdesc`, available since Yodl 3.05, can be used to defines an item *and* its description, using its suggested format (i.e., indenting the description relative to the item).

4.2.27 eit()

Indicates an item in an enumerated list. The `eit` macro should be used as an argument in `enumeration` macros.

4.2.28 ellipsis()

Sets ellipsis (...).

4.2.29 em(text)

Sets `text` as emphasized, usually italics.

4.2.30 email(address)

In HTML, this macro sets the `address` in a <`a href="mailto=.."`> locator. In other output formats, the `address` is sent to the output. The `email` macro is a special case of `url`.

4.2.31 enumeration(list)

`enumeration()` starts an enumerated list. Use `eit()` in the list to indicate items in the list.

4.2.32 euro()

Sets the euro currency symbol in latex, html, (and possibly sgml and xml). In all other conversions EUR which is the official textual abbreviation (cf. `http://ec.europa.eu/euro/entry.html`) is written. Note that LaTeX may require latexpackage()(eurosym).

4.2.33 evalsymbol(symbol)(expression)

Symbol *symbol* receives the value resulting from evaluating *expression*. E.g., if `sym` is a defined symbol, then

```
evalsymbol(sym)(SUBSTR(hello world)(3)(2))
```

assigns the value `lo` to `sym`.

4.2.34 fig(label)

This macro is a shorthand for `figure ref(label)` and just makes the typing shorter, as in `see fig(schematic) for ..` See `getfigurestring()` and `setfigurestring()` for the `figure` text.

4.2.35 figure(file)(caption)(label)

Sets the picture in `file` as a figure in the current document, using the descriptive text `caption`. The `label` is defined as a placeholder for the figure number and can be used in a corresponding `ref` statement. Note that the `file` must be the filename without extension: By default, Yodl will supply `.gif` when in HTML mode, or `.ps` when in LaTeX mode. Figures in other modes may not (yet) haven been implemented.

When converting to html, this macro uses three attribute-strings (if available). The string pushed first using an attrib-call defines the attributes for its <`figcaption`>

html-markup; the string pushed next defines the attributes for its <img> html-markup; the string pushed last defines the attributes for its <figure> html-markup. The figure macro's html output is organized like this:

```
<figure -attrib-string pushed last (if any)>
    <img ... -attrib-string pushed last but one>
    <figcaption -attrib-string pushed 2nd to last>
        ...
    </figcaption>
</figure>
```

Starting with Yodl 3.07.00 no alt="Figure # is shown here..." attribute is defined anymore for the img markup: an alt-attribute can easily be defined at the last attrib-call, using getfigurestring() to obtain Figure or its language-specific translation, and COUNTERVALUE(XXfigurecounter) to obtain the order-number of the figure shown in the next figure-macro call.

4.2.36 file(text)

Sets text as filename, usually boldface. In html attrib macro applies to the tag.

4.2.37 findex()

Generate an index entry for LaTex() or texinfo f-indices. Its argument is the index entry. See also the [cptv]index macro.

4.2.38 footnote(text)

Sets text as a footnote, or between parentheses when the output format does not allow footnotes.

4.2.39 gagmacrowarning(name name ...)

Prevents the yodl program from printing *cannot expand possible user macro*. E.g., if you have in your document the file(s) are .. then you might want to put before that: gagmacrowarning(file). Calls NOUSERMACRO.

4.2.40 getaffilstring()

Expands to the string that defines the name of *Affiliation Information*, by default *AFFILIATION INFORMATION*. Can be redefined for national language support by setaffilstring(). Currently, it is relevant only for txt.

4.2.41 getauthorstring()

Expands to the string that defines the name of *Author Information*, by default *AUTHOR INFORMATION*. Can be redefined for national language support by `setauthorstring()`. Currently, it is relevant only for txt.

4.2.42 getchapterstring()

Expands to the string that defines a 'chapter' entry, by default *Chapter*. Can be redefined for national language support by `setchapterstring()`.

4.2.43 getdatestring()

Expands to the string that defines the name of *Date Information*, by default *DATE INFORMATION*. Can be redefined for national language support by `setdatestring()`. Currently, it is relevant only for txt.

4.2.44 getfigurestring()

Returns the string that defines a 'figure' text, in captions or in the `fig()` macro. The string can be redefined using the `setfiguretext()` macro.

4.2.45 getpartstring()

Expands to the string that defines a 'part' entry, by default *Part*. Can be redefined for national language support by `setpartstring()`.

4.2.46 gettitlestring()

Expands to the string that defines the name of *Title Information*, by default *TITLE INFORMATION*. Can be redefined for national language support by `settitlestring()`. Currently, it is relevant only for txt.

4.2.47 gettocstring()

Expands to the string that defines the name of the table of contents, by default *Table of Contents*. Can be redefined for national language support by `settocstring()`.

4.2.48 htmlcommand(cmd)

Writes `cmd` to the output when converting to html. The `cmd` is not further expanded by Yodl.

4.2.49 htmlheadfile(file)

Adds the contents of `file` to the `head` section of an HTML document. The contents of file are not interpreted and should contain plain html text. This option can be useful when large bodies of text, like the contents of <`script`> sections, must be included into the head section of html documents. This macro is only active in the preamble, should only specified once, and is only interpreted for html conversions.

4.2.50 htmlheadopt(option)

Adds the literal text `option` to the current information in the `head` section of an HTML document. `Option` may (or: should) contain plain html text. A commonly occurring head option is `link`, defining, e.g., a style sheet. Since that option is frequently used, it has received a dedicated macro: `htmlstylesheet`. When large bodies of html-text must be added to html documents the macro `htmlheadfile` should be used. This macro is only active in the preamble and is only interpreted for html conversions.

4.2.51 htmlnewfile()

In HTML output, starts a new file. All other formats are not affected. Note that you must take your own provisions to access the new file; say via links. Also, it's safe to start a new file just befoore opening a new section, since sections are accessible from the clickable table of contents. The HTML converter normally only starts new files prior to a `chapter` definition.

4.2.52 htmlstyle(tag)(definition)

Adds a <`style type="text/css"`> `...` </`style`> element to the head section of an HTML document.
Use `htmlstyle` to specify one or more CSS definitions which are eventually inserted at the ellipsis (`...`) in the generic `style` definition shown above. E.g., (using `#rrggbb` to specify a color, where `rr` are two hexadecimal digits specifying the color's red component, `gg` two hexadecimal digits specifying the color's green component, and `bb` two hexadecimal digits specifying the color's blue component) specifying

```
htmlstyle(body)(color: #rrggbb; background-color: #rrggbb)
htmlstyle(h1)(color: blue; text-align: center)
htmlstyle(h2)(color: green)
```

results in the element

```
<style type="text/css">
    body {color: #rrggbb; background-color: #rrggbb;}
    h1 {color: blue; text-align: center;}
    h2 {color: green;}
```

```
</style>
```

The macros `htmlheadopt` and `htmlstylesheet` could also be used to put information into the head-section of an HTML document, but `htmlheadopt` is of a much more general nature, while `htmlstylesheet` refers to CSS elements stored in an external file. The macro `attrib` can be used to define *inline styles*.

The `htmlstyle` macro is only active in the preamble and is only interpreted for html conversions.
Refer to available CSS specifications (cf., `http://www.w3schools.com/cssref/` for an overview of how CSS specifications are used, and which CSS specifications are available).

By default the internal style specification
`figure {text-align: center;} img {vertical-align: center;}`
is used. If this is not appropriate, specify `nohtmlimgstyle()` in the preamble.

4.2.53 htmlstylesheet(url)

Adds a <`link rel="stylesheet" type="text/css" ...`> element to the head section of an HTML document, using `url` in its `href` field. The argument `url` is not expanded, and should be plain HTML text, without surrounding quotes. The macro `htmlheadopt` can also be used to put information in the head-section of an HTML document, but `htmlheadopt` is of a much more general nature. This macro is only active in the preamble and is only interpreted for html conversions.

4.2.54 htmltag(tagname)(start)

Sets `tagname` as a HTML tag, enclosed by < and >. When `start` is zero, the `tagname` is prefixed with `/`. As not all html tags are available through predefined Yodl-macros (there are too many of them, some are used very infrequently, and you can easily define macros for the tags for which Yodl doesn't offer predefined ones), the `htmltag` macro can be used to handle your own set of macros. In html the `attrib` macro is supported. E.g.,

```
attrib(title="World Health Organization")\
    htmltag(abbr)()WHO+htmltag(abbr)(0)
```

4.2.55 ifnewparagraph(truelist)(falselist)

The macro `ifnewparagraph` should be called from the `PARAGRAPH` macro, if defined. It will insert `truelist` if a new paragraph is inserted, otherwise `falselist` is inserted (e.g., following two consecutive calls of PARAGRAPH). This macro can be used to prevent outputting multiple blank lines.

4.2.56 includefile(file)

Includes `file`. The default extension `.yo` is supplied if necessary.

Since Yodl version 3.00.0 Yodl's default file inclusion behavior has changed. The current working directory no longer remains fixed at the directory in which Yodl is called, but is volatile, changing to the directory in which a yodl-file is located. This has the advantage that Yodl's file inclusion behavior now matches the way **C**'s `#include` directive operates. The originally implemented file inclusion behavior is used when Yodl's `-L` (`+NOTRANS(-{}-{})legacy-include`) option is used.

4.2.57 includeverbatim(file)

Include `file` into the output. No processing is done, `file` should be in preformatted form, e.g.:

```
whenhtml(includeverbatim(foo.html))
```

4.2.58 it()

Indicates an item in an itemized list. Items in `it` macros are arguments of `itemization` macros.

4.2.59 itdesc(itemname)(contents)

Starts an item and its description in a description list. Its name is `itemname`, the contents of the item is defined by `contents`. The `itemname` is defined by using the `dit` macro.

With `html` conversions the contents are surrounded by <dd> and </dd> tags, resulting in contents which are indented relative to the itemname. When the `attrib` macro is used it is applied to the itemname (`dt`-tags).

With other conversions the `contents` are quoted (as if using `quote(contents)`).

4.2.60 itemization(list)

Sets `list` as an itemizationd list. Use `it()` to indicate items in the list.

4.2.61 kindex()

Generate an index entry for LaTex() or texinfo k-indices. Its argument is the index entry. See also the `[cfptv]vindex` macro.

4.2.62 label(labelname)

Defines `labelname` as an anchor for a `link` command, or to stand for the last numbering of a section or figure in a `ref` command.

4.2.63 langle()

Character <

4.2.64 languagedutch()

Defines the Dutch-language specific headers. Active this macro via setlanguage(dutch).

4.2.65 languageenglish()

Defines the English-language specific headers. Active this macro via setlanguage(english).

4.2.66 languageportugese()

Defines the Portugese-language specific headers. Active this macro via setlanguage(portugese).

4.2.67 LaTeX()

The LaTeX symbol.

4.2.68 latexaddlayout(arg)

This macro is provided to add Yodl-interpreted text to your own LaTeX layout commands. The command is terminated with an end-of-line. See also the macro `latexlayoutcmds()`

4.2.69 latexcommand(cmd)

Writes `cmd` plus a white space to the output when converting to LaTeX. The `cmd` is not further expanded by Yodl.

4.2.70 latexdocumentclass(class)

Forces the LaTeX `\documentclass{...}` setting to `class`. Normally the class is defined by the macros `article`, `report` or `book`. This macro is an escape route in case you need to specify your own document class for LaTeX. This option is a *modifier* and must appear *before* the `article`, `report` or `book` macros.

4.2.71 latexlayoutcmds(NOTRANSs)

This macro is provided in case you want to put your own LaTeX layout commands into LaTeX output. The `NOTRANSs` are pasted right after the `\documentclass` stanza. The default is, of course, no local LaTeX commands. Note that this macro **does not** overrule my favorite LaTeX layout. Use `nosloppyhfuzz()` and `standardlayout()` to disable my favorite LaTeX layout.

4.2.72 latexoptions(options)

Set latex options: `documentclass[options]`. This command **must** appear before the document type is stated by `article`, `report`, etc..

4.2.73 latexpackage(options)(name)

Include latex package(s), a useful package is, e.g., `epsf`. This command **must** appear before the document type is stated by `article`, `report`, etc..

4.2.74 lchapter(label)(title)

Starts a new chapter in books or reports, setting a label at the beginning of the chapter.

4.2.75 letter(language)(date)(subject)(opening)(salutation)(author)

Starts a letter written in the indicated language. The date of the letter is set to 'date', the subject of the letter will be 'subject'. The letter starts with 'opening'. It is based on the 'letter.cls' document class definition. The macro is available for LaTeX only. Preamble command suggestions:

- `latexoptions(11pt)`
- `PageEnlarged(2cm)(2cm)(1.5cm)(3cm)`
- `letterreplyto(name)(address)(postalcode/city)`
- `letterfootitem(phone)(number)`, maybe e-mail too.
- `letteradmin(yourdate)(yourref)`
- `letterto(addressitem)`. Use a separate `letterto()` macro call for each new line of the address.

4.2.76 letteraddenda(type)(value)

Adds an addendum at the end of a letter. 'type' should be 'bijlagen', 'cc' or 'ps'.

4.2.77 letteradmin(yourdate)(yourref)

Puts 'yourletterfrom' and 'yourreference' elements in the letter. If left empty, two dashes are inserted.

4.2.78 letterfootitem(name)(value)

Puts a footer at the bottom of letter-pages. Up to three will usually fit. LaTeX only.

4.2.79 letterreplyto(name)(address)(zip city)

Defines the 'reply to' address in LaTeX or txt-letters.

4.2.80 letterto(element)

Adds 'element' as an additional line to the address in LaTeX letters.

4.2.81 link(description)(labelname)

In HTML output a clickable link with the text `description` is created that points to the place where `labelname` is defined using the `label` macro, and `attrib` macro applies to the <a> tag. Using `link` is similar to `url`, except that a hyperlink is set pointing to a location in the same document. For output formats other than HTML, only the `description` appears.

4.2.82 lref(description)(labelname)

This macro is a combination of the `ref` and `link` macros. In HTML output a clickable link with the text `description` and the label value is created that points to the place where `labelname` is defined using the `label` macro, and `attrib` macro applies to the <a> tag. For output formats other than HTML, only the `description` and the label value appears.

4.2.83 lsect(label)(title)

Starts a new section, setting a label at the beginning of the section. In html `attrib` macro applies to the <h2> tag.

4.2.84 lsubsect(label)(title)

Starts a new subsection. Other sectioning commands are `subsubsect` and `subsubsubsect`. A label is added just before the subsection. In html `attrib` macro applies to the <h3> tag.

4.2.85 lsubsubsect(label)(title)

Starts a sub-subsection, a label is added just before the section In html `attrib` macro applies to the <`h4`> tag.

4.2.86 lsubsubsubsect(label)(title)

Starts a sub-sub-sub section. This level of sectioning is not numbered, in contrast to 'higher' sectionings. A label is added just before the subsubsubection.

4.2.87 lurl(locator)

An url described by its Locator. For small urls with readable addresses.

4.2.88 mailto(address)

Defines the default `mailto` address for HTML output. Must appear **before** the document type is stated by `article`, `report`, etc..

4.2.89 makeindex()

Make index for latex.

4.2.90 mancommand(cmd)

Writes `cmd` to the output when converting to man. The `cmd` is not further expanded by Yodl.

4.2.91 manpage(title)(section)(date)(source)(manual)

Starts a manual page document. The `section` argument must be a number, stating to which section the manpage belongs to. Most often used are commands (1), file formats (5) and macro packages (7). The sectioning commands in a manpage are **not** `(n)sect` etc., but `manpage...()`. The first section **must** be the `manpagename`, the last section **must** be the `manpageauthor`. The standard manpage for section 1 contains the following sections (in the given order): `manpagename`, `manpagesynopsis`, `manpagedescription`, `manpageoptions`, `manpagefiles`, `manpageseealso`, `manpagediagnostics`, `manpagebugs`, `manpageauthor`. Optional extra sections can be added with `manpagesection`. Standard manpageframes for several manpagesections are provided in `/usr/local/share/yodl/manframes`. `Yodl` manual pages can be converted to `groff, html`, or plain ascii text formats.

4.2.92 manpageauthor()

Starts the AUTHOR entry in a `manpage` document. Must be the last section of a `manpage`.

4.2.93 manpagebugs()

Starts the BUGS entry in a `manpage` document.

4.2.94 manpagedescription()

Starts the DESCRIPTION entry in a `manpage` document.

4.2.95 manpagediagnostics()

Starts the DIAGNOSTICS entry in a `manpage` document.

4.2.96 manpagefiles()

Starts the FILES entry in a `manpage` document.

4.2.97 manpagename(name)(short description)

Starts the NAME entry in a `manpage` document. The short description is used by, e.g., the `whatis` database.

4.2.98 manpageoptions()

Starts the OPTIONS entry in a `manpage` document.

4.2.99 manpagesection(SECTIONNAME)

Inserts a non-required section named `SECTIONNAME` in a `manpage` document. This macro can be used to augment 'standard' manual pages with extra sections, e.g., EXAMPLES. **Note that** the name of the extra section should appear in upper case, which is consistent with the normal typesetting of manual pages.

4.2.100 manpageseealso()

Starts the SEE ALSO entry in a `manpage` document.

4.2.101 manpagesynopsis()

Starts the SYNOPSIS entry in a `manpage` document.

4.2.102 manttquoted(onOff)

With man-conversions arguments of tt macros are displayed as normal text. To enhance their visibility arguments of tt macros may be quoted, in which case they are surrounded by backtics and normal quotes. By default arguments of tt macros are not quoted. Quotation is used after calling `manttquoted(1)`, and is suppressed after calling `manttquoted(0)`. The macro is only active when converting to man.

4.2.103 mbox()

Unbreakable box in LaTeX. Other formats may have different opitions on our unbreakable boxex.

4.2.104 metaC(text)

Put a line comment in the output.

4.2.105 metaCOMMENT(text)

Write format-specific comment to the output.

4.2.106 mscommand(cmd)

Writes `cmd` to the output when converting to ms. The `cmd` is not further expanded by Yodl.

4.2.107 nbsp(count)

Inserts `count` 'non-breaking space' characters into the generated output; i.e., the space character is not optimized away. If the argument list is empty one non-breaking space is inserted.

4.2.108 nchapter(title)

Starts a chapter (in a `book` or `report`) without generating a number before the title and without placing an entry for the chapter in the table of contents. In html `attrib` macro applies to the <`h1`> tag.

4.2.109 nemail(name)(address)

Named email. A more consistent naming for url, lurl, email and nemail would be nice.

4.2.110 nl()

Forces a newline; i.e., breaks the current line in two.

4.2.111 nodeprefix(text)

Prepend text to node names, e.g.

```
nodeprefix(LilyPond) sect(Overview)
```

Currently used in texinfo descriptions only.

4.2.112 nodeprefix(text)

Prepend text to node names, e.g.

```
nodeprefix(LilyPond) sect(Overview)
```

Currently used in texinfo descriptions only.

4.2.113 nodetext(text)

Use text as description for the next node, e.g.

```
nodetext(The GNU Music Typesetter)chapter(LilyPond)
```

Currently used in texinfo descriptions only.

4.2.114 nohtmlfive()

Starting yodl 3.05 html-conversions by default use html5. This can be suppressed (in favor of using html4) by calling this macro. This macro merely suppresses writing the initial <`!DOCTYPE html`> to generated html files; it is only active in the preamble and is only interpreted for html conversions.

4.2.115 nohtmlimgstyle()

By default html-pages specify
(<`style type="text/css" img {vertical-align: bottom;}`></`style`>)
This macro suppresses this `img` CSS style specification. This macro is only active in the preamble and is only interpreted for html conversions.

4.2.116 nop(text)

Expand to text, to avoid spaces before macros e.g.: a^2. Although a+sups(2) should have the same effect.

4.2.117 nosloppyhfuzz()

By default, LaTeX output contains commands that cause it to shut up about hboxes that are less than 4pt overfull. When `nosloppyhfuzz()` appears before stating the document type, LaTeX complaints are 'vanilla'.

4.2.118 notableofcontents()

Prevents the generation of a table of contents. This is default in, e.g., `manpage` and `plainhtml` documents. When present, this option **must** appear before stating the document type with `article`, `report` etc..

4.2.119 notitleclearpage()

Prevents the generation of a `clearpage()` instruction after the typesetting of title information. This instruction is default in all non `article` documents. When present, must appear **before** stating the document type with `article`, `book` or `report`.

4.2.120 notocclearpage()

With the LaTeX converter, no `clearpage()` instruction is inserted immediately beyond the document's table of contents. The `clearpage()` instruction is default in all but the `article` document type. When present, must appear **before** stating the document type with `article`, `book` or `report`. With other converters than the LaTeX converter, it is ignored.

4.2.121 notransinclude(filename)

Reads filename and inserts it literally in the text not subject to macro expansion or character translation. No information is written either before or after the file's contents, not even a newline.

4.2.122 noxlatin()

When used in the preamble, the LaTeX converter disables the inclusion of the file `xlatin1.tex`. Normally this file gets included in the LateX output files to ensure the conversion of high ASCII characters (like é) to LaTeX-understandable codes. (The file `xlatin1.tex` comes with the `Yodl` distribution.)

4.2.123 nparagraph(title)

Starts a non-numbered paragraph (duh, corresponds to subparagraph in latex).

4.2.124 npart(title)

Starts a part in a `book` document, but without numbering it and without entering the title of the part in the table of contents. In html `attrib` macro applies to the <h1> tag.

4.2.125 nsect(title)

Starts a section, but does not generate a number before the `title` nor an entry in the table of contents. Further sectioning commands are `nsubsect`, `nsubsubsect` and `nsubsubsubsect`. In html `attrib` macro applies to the <h2> tag.

4.2.126 nsubsect(title)

Starts a non-numbered subsection. In html the `attrib` macro applies to the <h3> tag.

4.2.127 nsubsubsect(title)

Starts a non-numbered sub-sub section. In html `attrib` macro applies to the <p> tag.

4.2.128 nsubsubsect(title)

Starts a non-numbered sub-subsection.

4.2.129 PageEnlarged(up)(height)(left)(width)

Enlarges the usable width and height of printed pages: the first line printed on paper is moved up by `up` units, the usable paper height is enlarged by `height` units, the left margin is moved to the left by t(left) units, and the usable paper width is enlarged by (width) units. E.g., A4 pages are by default using relatively small text blocks. Instead the pages can be used more efficiently by specifying . his macro should be called in the preamble. The macro is available only for LaTeX conversions.

4.2.130 paragraph(title)

Starts a paragraph. This level of sectioning is not numbered, in contrast to 'higher' sectionings (duh, corresponds to subparagraph in latex). In html `attrib` macro applies to the <p> tag.

4.2.131 part(title)

Starts a new part in a `book` document. In html `attrib` macro applies to the <h1> tag.

4.2.132 pindex()

Generate an index entry for LaTex() or texinfo p-indices. Its argument is the index entry. See also the `[cftv]index` macro.

4.2.133 plainhtml(title)

Starts a document for only a plain HTML conversion. Not available in other output formats. Similar to `article`, except that an author- and date field are not needed.

4.2.134 printindex()

Make index for texinfo (?).

4.2.135 quote(text)

Sets the text as a quotation. Usually, the text is indented, depending on the output format. In html the `attrib` macro is recognized by the <`blockquote`> tag.

4.2.136 rangle()

Inserts the right angle character (>).

4.2.137 redef(macro)(nrofargs)(redefinition)

Defines macro `macro` to expand to `redefinition`. Similar to `def`, but any pre-existing definition is overruled. Use `ARG`x in the redefinition part to indicate where the arguments should be pasted. E.g., `ARG1` places the first argument, `ARG2` the second argument, etc...

4.2.138 redefinemacro(macro)(nrofargs)(redefinition)

Defines macro `macro` to expand to `redefinition`. Similar to `def`, but any pre-existing definition is overruled. Use `ARG`x in the redefinition part to indicate where the arguments should be pasted. E.g., `ARG1` places the first argument, `ARG2` the second argument, etc... This commands is actually calling redef().

4.2.139 ref(labelname)

Sets the reference for `labelname`. Use `label` to define a label.

4.2.140 report(title)(author)(date)

Starts a report type document. The top-level sectioning command in a report is `chapter`. In html the way the headings are displayed can be tuned using CSS id selector specifications: the title has `id="title"`, the author `id="author"`, and the date `id="date"`.

4.2.141 roffcmd(dotcmd)(sameline)(secondline)(thirdline)

Sets a t/nroff command that starts with a dot, on its own line. The arguments are: `dotcmd` - the command itself, e.g., `.IP`; `sameline` - when not empty, `sameline` is set on the same line, following the `dotcmd`; `secondline` - when not empty, is set on the next line; `thirdline` - when not empty, is set on the third line. Note: `dotcmd` and `thirdline` are not further expanded by `Yodl`, the other arguments are.

4.2.142 row(contents)

The argument `contents` may contain a man-page alignment specification (only one specification can be entered per row), using `setmanalign()`. If omitted, the standard alignment is used. Furthermore it contains the contents of the elements of the row, using `cell()` or `cells()` macros. If `cells()` is used, `setmanalign()` should have been used too. In this macro call only the `cell()`, `cells()` and `setmanalign()` macros should be called. Any other macro call may produce unexpected results.

The `row` macro defines a counter `XXcol` that can be inspected and is incremented by predefined macros adding columns to a row. The counter is initially 0. Predefined macros adding columns to a row add the number of columns they add to the row inserting the contents of those columns. These macros rely on the correct value of this counter and any user-defined macros adding columns to table rows should correctly update `XXcol`. In html `attrib` macro applies to the <tr> tag.

Instead of using `row` in `table`, consider using `tr` in `tbl`.

4.2.143 rowline()

Sets a horizontal line over the full width of the table. See also the `columnline` macro. Use `rowline` instead of a `row` macro call to provide a table with a horizontal line-separator.

Instead of using `rowline` consider using `tline` in a `tbl` argument.

4.2.144 sc(text)

Set `text` in the tt (code) font, using small caps. In html the `attrib` macro is not supported, while the code section is embedded in a <`div style="font-size: 90%"`> section.

4.2.145 sect(title)

Starts a new section. In html the `attrib` macro is recognized by the <h2> tag.

4.2.146 setaffilstring(name)

Defines `name` as the 'affiliation information' string, by default *AFFILIATION INFORMATION*. E.g., after `setaffilstring(AFILIACION)`, `Yodl` outputs this Spanish string to describe the affiliation information. Currently, it is relevant only for txt.

4.2.147 setauthorstring(name)

Defines `name` as the 'Author information' string, by default *AUTHOR INFORMATION*. E.g., after `setauthorstring(AUTOR)`, `Yodl` outputs this portuguese string to describe the author information. Currently, it is relevant only for txt.

4.2.148 setchapterstring(name)

Defines `name` as the 'chapter' string, by default *Chapter*. E.g., after `setchapterstring(Hoofdstuk)`, `Yodl` gains some measure of national language support for Dutch. Note that LaTeX support has its own NLS, this macro doesn't affect the way LaTeX output looks.

4.2.149 setdatestring(name)

Defines `name` as the 'date information' string, by default *DATE INFORMATION*. E.g., after `setdatestring(DATA)`, `Yodl` outputs this portuguese string to describe the date information. Currently, it is relevant only for txt.

4.2.150 setfigureext(name)

Defines the `name` as the 'figure' extension. The extension should include the period, if used. E.g., use setfigureext(.ps) if the extensions of the figure-images should end in `.ps`

4.2.151 setfigurestring(name)

Defines the `name` as the 'figure' text, used e.g. in figure captions. E.g., after `setfigurestring(Figuur)`, Yodl uses Dutch names for figures.

4.2.152 sethtmlfigureext(ext)

Defines the filename extension for HTML figures, defaults to `.jpg`. Note that a leading dot must be included in `ext`. The new extension takes effect starting with

the following usage of the `figure` macro. It is only active in html, but otherwise acts identically as setfigureext().

See also the `setlatexfigureext` macro.

4.2.153 htmlmetacharset(meta-charset)

Adds <`meta charset="meta-charset"`> to the head of html documents. By default <`meta charset="UTF-8"`> is used. This macro is only active in the preamble and is only interpreted for html conversions.

4.2.154 setincludepath(name)

Sets a new value of the include-path specification used when opening .yo files. A warning is issued when the path specification does not include a .: element. Note that the local directory may still be an element of the new include path, as the local directory may be the only or the last element of the specification. For these eventualities the new path specification is not checked.

4.2.155 setlanguage(name)

Installs the headers specific to a language. The argument must be the name of a language, whose headers have been set by a corresponding languageXXX() call. For example: languagedutch(). The language macros should set the names of the headers of the following elements: table of contents, affiliation, author, chapter, date, figure, part and title

4.2.156 setlatexalign(alignment)

This macro defines the table alignment used when setting tables in LaTeX. Use as many `l` (for left-alignment), `r` (for right alignment), and `c` (for centered-alignment) characters as there are columns in the table. See also `table()`.

Instead of using the `table` macro consider using the `tbl` macro.

4.2.157 setlatexfigureext(ext)

Defines the filename extension for encapsulated PostScript figures in LaTeX, defaults to `.ps`. The dot must be included in t new extension `ext`. The new extension takes effect starting with a following usage of the `figure` macro. It is only active in LaTeX, but otherwise acts identically as setfigureext().

See also the `sethtmlfigureext` macro.

4.2.158 setlatexverbchar(char)

Set the char used to quote LaTeX `\verb` sequences

4.2.159 setmanalign(alignment)

This macro defines the table alignment in the context of the `table` macro, and is used when setting tables in man-pages (see **tbl**(1)).

Use as many `l` (for left-alignment), `r` (for right alignment), and `c` (for centered-alignment) characters as there are columns in the table.

Use `s` to indicate that the column to its left is combined (spans into) the current column. Use this specification when cells spanning multiple columns must be defined.

Each row in a table which must be convertible to a manpage may be preceded by its own `setmanalign` call.

Note that neither `rowline` nor `columnline` requires `setmanalign` specifications, as these macros define rows by themselves. It is the responsibility of the author to ensure that the number of alignment characters is equal to the number of columns of the table.

Instead of using the `table` macro consider using the `tbl` macro.

4.2.160 setpartstring(name)

Defines `name` as the 'part' string, by default *Part*. E.g., after `setpartstring(Teil)`, Yodl identifies parts in the German way. Note that LaTeX output does its own national language support; this macro doesn't affect the way LaTeX output looks.

4.2.161 setrofftab(x)

Sets the character separating items in a line of input data of a `roff` (manpage) table. By default it is set to `~`. This separator is used internally, and needs only be changed (into some unique character) if the table elements themselves contain `~` characters.

This macro can be used in the context of the `table` and `tbl` macros.

4.2.162 setrofftableoptions(optionlist)

Set options used for man-conversion, as used by the `tbl` and `table` macros. By default no options are used. Multiple options should be separated by blanks. From the **tbl**(1) manpage, the following options are available:

- `center` Centers the table; default is left-justified. In the context of the `tbl` macro this is implied when the `tbl` macro is specified as argument of the `center` macro.
- `expand` Makes the table as wide as the current line length
- `box` Encloses the table in a box
- `allbox` Encloses each item of the table in a box

See also `setrofftab` which is used to set the character separating items in a line of input data.

4.2.163 settitlestring(name)

Defines `name` as the 'title information' string, by default *TITLE INFORMATION*. E.g., after `settitlestring(TITEL)`, `Yodl` outputs this Dutch string to describe the title information. Currently, it is relevant only for txt.

4.2.164 settocstring(name)

Defines `name` as the 'table of contents' string, by default *Table of Contents*. E.g., after `settocstring(Inhalt)`, `Yodl` identifies the table of contents in the German way. Note that LaTeX output does its own national language support; this macro doesn't affect the way LaTeX output looks.

4.2.165 sgmlcommand(cmd)

Writes `cmd` to the output when converting to sgml. The `cmd` is not further expanded by Yodl.

4.2.166 sgmltag(tag)(onoff)

Similar to `htmltag`, but used in the SGML converter.

4.2.167 sloppyhfuzz(points)

By default, LaTeX output contains commands that cause it to shut up about hboxes that are less than 4pt overfull. When `sloppyhfuzz()` appears before stating the document type, LaTeX complaints occur only if hboxes are overfull by more than `points`.

4.2.168 standardlayout()

Enables the default LaTeX layout. When this macro is absent, then the first lines of paragraphs are not indented and the space between paragraphs is somewhat larger. The `standardlayout()` directive must appear **before** stating the document type as `article`, `report`, etc..

4.2.169 strong(contents)

In html and xml the `contents` are set between <`strong`> and </`strong`> tags. In html `attrib` macro applies to the <`strong`> tag.

4.2.170 subs(text)

Sets text in subscript in supporting formats. In html the `attrib` macro is recognized by the <`sub`> tag.

For *super*scripting, the `sups` macro can be used.

4.2.171 subsect(title)

Starts a new subsection. Other sectioning commands are `subsubsect` and `subsubsubsect`. In html `attrib` macro applies to the <`h3`> tag.

4.2.172 subsubsect(title)

Starts a sub-subsection. In html `attrib` macro applies to the <`h4`> tag.

4.2.173 subsubsubsect(title)

Starts a sub-sub-sub-subsection. This level of sectioning is not numbered, in contrast to 'higher' sectionings.

4.2.174 sups(text)

Sets text in superscript in supporting formats In html the `attrib` macro is recognized by the <`sup`> tag.

For *sub*scripting, the `subs` macro can be used.

4.2.175 table(nColumns)(alignment)(Contents)

Instead of using the `table` macro, consider using the `tbl` macro.

The `table()`-macro defines a table. Its first argument specifies the number of columns in the table.

Its second argument specifies the (standard) alignment of the information within the cells as used by LaTeX or man/ms. Use `l` for left-alignment, `c` for centered-alignment and `r` for right alignment.

Its third argument defines the contents of the table which are the rows, each containing column-specifications and optionally man/ms alignment definitions for this row.

See also the `tableatt` macro and the specialized `setmanalign()` macro.

4.2.176 tableatt(nColumns)(alignment)(Contents)

Instead of using the `tableatt` macro, consider using the `tbl` macro.

The `tableatt()`-macro defines a table. The last `attrib` call that was specified before using the `tableatt()`-macro is used to specify html attributes for the table. E.g., to center a table in html use

```
attrib(style="margin-left:auto;margin-right:auto;")
tableatt(...)
```

The macro's first argument specifies the number of columns in the table. Its second argument specifies the (standard) alignment of the information within the cells as used by LaTeX or man/ms. Use `l` for left-alignment, `c` for centered-alignment and `r` for right alignment. Its third argument defines the contents of the table which are the rows, each containing column-specifications and optionally man/ms alignment definitions for this row.

See also the `table` macro and the specialized `setmanalign()` macro.

4.2.177 tac(alignment)(contents)

This macro is one of the four macros that can be used to define column contents of rows of tables defined by the `tbl` macro. Alternatively, `tc, tnc`, and `tnac` can be used.

The `tac` macro is used as argument of the `tr` macro. The cell's alignment is defined by the `alignment` specification, containing at most two alignment specifications: a horizontal alignment (one of c, l, r (centered, left-aligned, right-aligned)) and a vertical alignment (one of t, b (vertical top- and bottom-alignment)) (not all conversion types may support all alignment types, e.g., man-conversion does not support vertical bottom alignment). Specifications other than c, l, r, b, and t and specifications beyond the second one are ignored. The result of specifying conflicting alignment types (e.g., `lr` or `tb`) is not defined.

When converting to `man`, if the table's contents should span multiple rows, then a **groff**/**troff**(1) *text block* must be used. Since most tables do not use this, a text block is not generated by default. To actually wrap the contents of column elements in a text block while converting to `man` prefix the first text block requiring a text-block wrapping by `twrap(1)`, and end the last text block requiring a text-block wrapping by `wrap(0)`.

The `tao` macro can be used to overrule the specified alignment for a specific conversion type.

The macro `tac` recognizes `attrib`.

4.2.178 tao(type)(specification)

This macro is used inside the `tbl` macro to override the alignment specification that would otherwise be used for the next table element. It is only active for the next `tc, tnc, tac,` or `tnac` call. Its first argument defines the conversion type for which the override should be used, its second argument defines the alignment specification to use.

Here are some examples:

```
tbl(lr)(
    tr(
        tc(left aligned)
        tc(right aligned)
    )
    tr(
        tao(html)(c)
          tc(left aligned, centered with html)
        tao(latex)(l)
        tao(man)(l)
          tac(c)(centered, latex and man: left aligned)
    )
)
```

Further details about the `tao` macro are provided in the **yodltables**(7) man-page.

4.2.179 tbl(align)(contents)

The `tbl` macro refines the more basic `table` macro. It was named after the **tbl**(1) table formatting program used with **troff**(1) documents.

The `tbl` macro currently is available for `html, man/ms, latex` and `txt` conversions.

Its first argument defines the alignment of the information in the table's columns, and is used by all conversions except `txt`. Use `l` for left-alignment, `c` for centered-alignment and `r` for right-alignment. Individual cells of the table may override these default settings using the macros `tac` and `tnac`.

Its second argument defines the contents of the table consisting of rows (using `tr`), and horizontal lines (using `tline`), which may extend over the full table width or may cover one or more individual columns. With `txt` conversion rough approximations of horizontal lines are used.

Tables defined by the `tbl` macro are centered (pseudo centering (8 space characters) is used for `txt` conversion) when used as argument of the `center` macro.

See also the `tao` macro for information about how to realize specific alignments for specific conversion types.

When defining `tbl` tables it is advised to clearly layout the table specification. To avoid inadvertently introducing new lines lines should end in a backslash (or The macro `tbl` recognizes `attrib`.

4.2.180 tc(contents)

This macro is one of the four macros that can be used to define column contents of rows of tables defined by the `tbl` macro. Alternatively, `tnc, tac`, and `tnac` can be used.

The `tc` macro is used as argument of the `tr` macro. Its order within a row defines its type attribute, using the alignment specification defined by the first argument of the `tbl` call. E.g., if `tbl(clr)(...)` was used, then the contents of the first `tc` call in a `tr` is centered in the table's first column; the contents of the second `tc` call is left-aligned in the table's second column; and the contents of the third `tc` call is right-aligned in the table's third column.

When converting to `man`, if the table's contents should span multiple rows, then a **groff**/**troff**(1) *text block* must be used. Since most tables do not use this, a text block is not generated by default. To actually wrap the contents of column elements in a text block while converting to `man` prefix the first text block requiring a text-block wrapping by `twrap(1)`, and end the last text block requiring a text-block wrapping by `wrap(0)`.

The `tao` macro can be used to overrule the default alignment specification for a specific conversion type.

The macro `tc` recognizes `attrib`.

4.2.181 tcell(text)

Roff helper to set a table text-cell, i.e., a paragraph. For LaTeX special table formatting p{} should be used.

When using the `tbl` macro for defining tables the `twrap` macro can be used to set table elements in text-blocks (i.e., enclosing text, possibly containing newlines in `T{` and `T}` sequences).

4.2.182 telycommand(cmd)

Writes `cmd` to the output when converting to tely. The `cmd` is not further expanded by Yodl.

4.2.183 TeX()

The TeX symbol.

4.2.184 texinfocommand(cmd)

Writes `cmd` to the output when converting to texinfo. The `cmd` is not further expanded by Yodl.

4.2.185 tindex()

Generate an index entry for LaTex() or texinfo t-indices. Its argument is the index entry. See also the `[cfpv]index` macro.

4.2.186 titleclearpage()

Forces a new page (using `clearpage`) following the title of a document. This is already the default in books and reports, but can be overruled using `notitleclearpage`. When present, it must appear in the *preamble*, i.e., before the document type is stated as article, book or report.

4.2.187 tline(beginNr)(endNr)

This macro is used to insert a horizontal line spanning one or more columns of a table defined by the `tbl` macro.

If `endNr` is not specified, and the `tline` call does not follow a previous `line` call in which `endNr` *was* specified, then a horizontal line spanning the full width of the table is defined (except when converting to plain text in which case a line of `beginNr` - (minus) characters is written; if `beginNr` is not specified then a line of 60 - characters is written).

If `endNr` is not specified, but the `tline` call follows previous `tline` calls that did specify `endNr` then the current row is ended.

If `endNr` is specified, then a horizontal line is set, starting at column number `beginNr` continuing through column number `endNr`. Note that these are *numbers*, not *offsets*: `beginNr` should be at least 1, `endNr` must at least be equal to `beginNr` and should be at most equal to the number of columns in the table. The `beginNr` values of subsequent `tline` calls refer to the same row as the first `tline` call, and must exceed `endNr` of the previous `tline` call.

Examples:

```
tline()()       sets a line spanning the full table width
tline(1)(1)     sets a line in column 1
tline(3)(4)     sets another line in column 3 and 4
tline()()       ends the previous line
tline()()       sets a line spanning the full table width
```

4.2.188 tnac(nCells)(alignment)(contents)

This macro is one of the four macros that can be used to define column contents of rows of tables defined by the `tbl` macro. Alternatively, `tc, tac`, and `tnc` can be used.

The `tnac` macro is used as argument of the `tr` macro. Its first argument defines the number of columns spanned by the contents (2nd argument) of the `tnc` macro.

The cell's alignment is defined by the macro's second (`alignment`) specification, containing at most two alignment specifications: a horizontal alignment (one of c, l, r (centered, left-aligned, right-aligned)) and a vertical alignment (one of t, b (vertical top- and bottom-alignment)) (not all conversion types may support all alignment types). Specifications other than c, l, r, b, and t and specifications beyond the

second one are ignored. The result of specifying conflicting alignment types (e.g., `lr` or `tb`) is not defined.

When converting to `man`, if the table's contents should span multiple rows, then a **groff**/**troff**(1) *text block* must be used. Since most tables do not use this, a text block is not generated by default. To actually wrap the contents of column elements in a text block while converting to `man` prefix the first text block requiring a text-block wrapping by `twrap(1)`, and end the last text block requiring a text-block wrapping by `wrap(0)`. Multiple rows in a text block are top-aligned with left and righ neighboring cells.

The `tao` macro can be used to overrule the default alignment specification for a specific conversion type.

The macro `tnac` recognizes `attrib`.

4.2.189 tnc(nCells)(contents)

This macro is one of the four macros that can be used to define the column contents of rows of tables that are defined by the `tbl` macro. Alternatively, `tc, tac`, and `tnac` can be used.

The `tnc` macro is used as argument to the `tr` macro. Its first argument defines the number of columns spanned by the contents (2nd argument) of the `tnc` macro. The contents are centered in the `nCells` columns.

When converting to `man`, if the table's contents should span multiple rows, then a **groff**/**troff**(1) *text block* must be used. Since most tables do not use this, a text block is not generated by default. To actually wrap the contents of column elements in a text block while converting to `man` prefix the first text block requiring a text-block wrapping by `twrap(1)`, and end the last text block requiring a text-block wrapping by `wrap(0)`. Multiple rows in a text block are top-aligned with left and righ neighboring cells.

The `tao` macro can be used to overrule the default alignment specification for a specific conversion type.

The macro `tc` recognizes `attrib`.

4.2.190 tocclearpage()

With the LaTeX converter, a `clearpage()` directive if inserted, immediately following the document's table of contents. This is already the default in all but the `article` document type, but it can be overruled by `notocclearpage()`. When present, it must appear in the *preamble*; i.e., before the document type is stated with `article`, `book` or `report`. With other converters than the LaTeX converter, it is ignored.

4.2.191 tr(contents)

This macro defines the rows of a table that is defined by the `tbl` macro.

The macro `tr` expects one argument: the contents of the row, defining the row's column elements. It is not used for defining a (partial) horizontal line: to set horizontal lines in a table defined by the `tbl` macro use the macro `tline`.

Normally the contents of the columns in a (`tr`) row is defined by of one or more calls to the macros `tc, tac, tnc,` and/or `tnac`.

The macro `tr` recognizes `attrib`.

4.2.192 tt(text)

Sets `text` in teletype font, and prevents it from being expanded. When converting to man, `text` is surrounded by a backtick and a single quote character.

For unbalanced parameter lists, use `CHAR(40)` to get `(` and `CHAR(41)` to get `)`.

The `tt` macro does interpret character tables as well as any `SUBST` definitions. This is usually what is intended. In situations where this is unwelcome the `ttbegin` and `ttend` pair of macros can be used, between which the builtin commands `PUSHSUBST, POPSUBST, NOEXPAND` and/or `NOTRANS` can be used. E.g., to clearly show two hyphens in LaTeX teletype font use

```
ttbegin()--ttend()
```

rather than

```
tt(--)
```

Likewise, use `ttbegin` and `ttend` if the teletype text contains accented letters like ë. To set this in teletype font use `ttbegin()\"e+ttend()`.

With html conversions the `attrib` macro applies to the <`code`> tag.

With man conversions the arguments of tt macros can be quoted. See the `manttquoted` macro for details.

4.2.193 ttbegin()

Initiates text set in teletype font. Following the text to set in teletype font a `ttend()` macro should be called.

Usually the `tt` macro can be used instead of the `ttbegin -- ttend` combination. However, `tt` interprets character tables as well as `SUBST` definitions. In situations where this is unwelcome the `ttbegin` and `ttend` pair of macros can be used, between which builtin commands like `PUSHSUBST, POPSUBST, NOEXPAND` and/or `NOTRANS` can be used.

In html the `attrib` macro applies to the <`code`> tag.

4.2.194 ttend()

Ends text set in teletype font following `ttbegin`. Refer to the `ttbegin` macro's description for details.

4.2.195 twrap(value)

Used for man/ms conversions only: when called with a non-zero argument before using the `tc, tnc, tac`, and `tnac` macros then their contents are wrapped in text blocks (`T{ ... T}` blocks). To end the wrapping `twrap(0)` must be called. E.g., in the following row-definition the contents of columns three and four are set in T-blocks when converting to man/ms:

```
tr(tc(one)tc(two)twrap(1)tc(one)tc(two)twrap(0))
```

4.2.196 txtcommand(cmd)

Writes `cmd` to the output when converting to txt. The `cmd` is not further expanded by Yodl.

4.2.197 url(description)(locator)

In LaTeX documents the `description` is sent to the output. For HTML, a link is created with the descriptive text `description` and pointing to `locator`. The `locator` should be the full URL, including service; e.g, `http://www.icce.rug.nl`, but excluding the double quotes that are necessary in plain HTML. Use the macro `link` to create links within the same document. For other formats, something like *description [locator]* will appear. In html `attrib` macro applies to the <a> tag.

4.2.198 verb(text)

Sets `text` in verbatim mode: not subject to macro expansion or character table expansion, and starting with Yodl version 4.00.00: not using `SUBST` definitions. See also `verborg`, which does not provide the protection against `SUBST` definitions.

While converting Yodl-documents to target document types Yodl frequently uses the (not further documented) builtin function `XXSUBST`. In the unlikely event that the text `XXSUBST(...)` must be written in a document, the sequence

```
XXSUBST+CHAR(40)...CHAR(41)
```

can be used.

The text that is passed as argument to the `verb`-macro appears literally on the output, usually in a teletype font (that depends on the output format). This macro is for larger chunks, e.g., listings.

When starting the `verb(` macro on a line by itself and writing the closing parenthesis on a line by itself then those newlines also appear in the resulting verbatim text. To omit those lines immediately start the first line of text after `verb` and append the closing parenthesis to the last line of verbatim text. E.g.,

```
verb(    First line of indented literal text
    Last line of indented literal text)
```

When unbalanced parameter lists are required, use `CHAR(40)` to get `(` and `CHAR(41)` to get `)`.

4.2.199 verbinclude(filename)

Reads filename and inserts it literally in the text, set in verbatim mode. not subject to macro expansion. The text appears literally on the output, usually in a teletype font (that depends on the output format). This macro is an alternative to `verb(...)`, when the text to set in verbatim mode is better kept in a separate file.

4.2.200 verbinsert(args)

Passes `args` to **yodlverbinsert**(1), inserting its output into the converted file. This macro can be used to insert, e.g., a line-numbered indented file, or a labeled subsection of a file, into the file that's currently being written by `yodl`. E.g,

```
verbinsert(-ans4 file)       -- inserts file, showing line
                                numbers, using a 4 blank-space
                                character wide indentation.

verbinsert(-ns4 //SECT file) -- inserts the section of file,
                                labeled //SECT file, showing
                                line numbers, using a 4
                                blank-space character wide
                                indentation.
```

4.2.201 verborg(text)

Sets `text` in verbatim mode: not subject to macro expansion or character table expansion, and starting with Yodl version 4.00.00: this macro replaces the previously defined `verb` macro. The current `verb` macro surrounds this macro by `PUSHSUBST` and `POPSUBST`.

The text that is passed as argument to the `verborg`-macro appears literally on the output, albeit that `SUBST` definitions are processed, usually in a teletype font (that depends on the output format). This macro is for larger chunks, e.g., listings.

When unbalanced parameter lists are required, use `CHAR(40)` to get `(` and `CHAR(41)` to get `)`.

4.2.202 verbpipe(command)(text)

Pipe text through command, but don't expand the output.

4.2.203 vindex()

Generate an index entry for LaTex() or texinfo v-indices. Its argument is the index entry. See also the `[cfkpt]index` macro.

4.2.204 whenhtml(text)

Sends `text` to the output when in HTML conversion mode. The text is further expanded if necessary.

4.2.205 whenlatex(text)

Sends `text` to the output when in LATEX conversion mode. The text is further expanded if necessary.

4.2.206 whenman(text)

Sends `text` to the output when in MAN conversion mode. The text is further expanded if necessary.

4.2.207 whenms(text)

Sends `text` to the output when in MS conversion mode. The text is further expanded if necessary.

4.2.208 whensgml(text)

Sends `text` to the output when in SGML conversion mode. The text is further expanded if necessary.

4.2.209 whentely(text)

Sends `text` to the output when in TELY conversion mode. The text is further expanded if necessary.

4.2.210 whentexinfo(text)

Sends `text` to the output when in TEXINFO conversion mode. The text is further expanded if necessary.

4.2.211 whentxt(text)

Sends `text` to the output when in TXT conversion mode. The text is further expanded if necessary.

4.2.212 whenxml(text)

Sends `text` to the output when in XML conversion mode. The text is further expanded if necessary.

4.2.213 xit(itemname)

Starts an xml menu item where the file to which the menu refers to is the argument of the xit() macro. It should be used as argument to xmlmenu(), which has a 3rd argument: the default path prefixed to the xit() elements.

This macro is only available within the xml-conversion mode. The argument must be a full filename, including .xml extension, if applicable.

No .xml extension indicates a subdirectory, containing another sub-menu.

4.2.214 xmlcommand(cmd)

Writes `cmd` to the output when converting to xml. The `cmd` is not further expanded by Yodl.

4.2.215 xmlmenu(order)(title)(menulist)

Starts an xmlmenu. Use itemization() to define the items. Only available in xml conversion. The menutitle appears in the menu as the heading of the menu. The menulist is a series of xit() elements, containing the name of the file to which the menu refers as their argument (including a final /). Prefixed to evert every xit()-element is the value of XXdocumentbase.

Order is the the 'order' of the menu. If omitted, no order is defined.

4.2.216 xmlnewfile()

In XML output, starts a new file. All other formats are not affected. Note that you must take your own provisions to access the new file; say via links. Also, it's safe to start a new file just befoore opening a new section, since sections are accessible from the clickable table of contents. The XML converter normally only starts new files prior to a `chapter` definition.

4.2.217 xmlsetdocumentbase(name)

Defines `name` as the XML document base. No default. Only interpreted with xml conversions. It is used with the figure and xmlmenu macros.

4.2.218 xmltag(tag)(onoff)

Similar to `htmltag`, but used in the XML converter.

4.3 Conversion-related topics

4.3.1 Conversion-type specific literal commands

According to the format of the output file, the macro package defines a given symbol:

- `latex` when the output format is LaTeX,
- `html` when the output format is HTML,
- `man` when the output format is groff in conjunction with the man macro package,
- `ms` when the output format is groff with the ms package,
- `sgml` when the output format is SGML,
- `txt` when the output format is plain ASCII.
- `xml` when the output format is XML.

The defined symbol can be tested in a document to determine the conversion type.

Furthermore, the package defines the following macros to send literal text (commands in the output format) to the output file:

- `latexcommand(cmd)`: sends the LaTeX command `cmd` when in LaTeX conversion mode. The `cmd` is not further expanded.
- `htmlcommand(cmd)`: sends the HTML command `cmd` when in HTML conversion mode. The `cmd` is not further expanded.
- `htmltag(tag)(onoff)`: sends <tag> to the output when `onoff` is nonzero, or sends </tag> when `onoff` is zero. Only active in HTML conversions.
- `mancommand(cmd)`: sends `cmd` to the output when in man conversion mode. The `cmd` is not further expanded.
- `mscommand(cmd)`: sends `cmd` to the output when in ms conversion mode. The `cmd` is not further expanded.

- `roffcmd(dotcmd)(trailer)(secondline)(thirdline)`: sends a command to the output when in `man` or `ms` conversion mode. The `dotcmd` is the typical `groff` command that starts with a dot. All other arguments may be empty, but when given are interpreted as follows. The `trailer` follows the `dotcmd` on the same line. The `secondline` is sent on a separate line following the `dotcmd` and `trailer`. The `thirdline` is sent after that. Of the four arguments, `dotcmd` and `thirdline` are **not** subject to further expansion. All other arguments are further expanded if necessary.

  The `roffcmd` macro illustrates the complexity of dot-commands for the divers `groff` macro packages. E.g., a section title for the `man` package should look as

  ```
  .SH "Section Title"
  ```

  while the same command for the `ms` macro package must be sent as

  ```
  .SH
  Section Title
  .PP
  ```

  The `roffcmd` macro can be used to send these commands to the output file as follows:

  ```
  COMMENT(For the man output format:)
  roffcmd(.SH)("Section Title")()()

  COMMENT(For the ms output format:)
  roffcmd(.SH)()(Section Title)(.PP)()
  ```

- `sgmlcommand(cmd)`: sends the SGML command `cmd` when in SGML conversion mode. The `cmd` is not further expanded.
- `sgmltag(tag)(onoff)`: sends <`tag`> when `onoff` is nonzero, or sends </`tag`> when `onoff` is zero. Only active in SGML conversions.
- `txtcommand(cmd)`: implemented for compatibility reasons, though a 'command' in plain ASCII output doesn't make much sense. The usefulness of this macro is rather in the fact that it only produces output when in ASCII conversion mode.

The above commands can be used to quickly implement a macro. E.g., the macro package implements the `it` macro (which starts an item in a list) as:

```
DEFINEMACRO(it)(0)(
    latexcommand(\item )
    htmlcommand(<li> )
    ....
)
```

Depending on the output format, `it()` applies one of the above expansions.

The above described *format*`command()` macros are implemented to send not further expanded strings (i.e., commands) to the output. The macro package also implements `when`*format*`()` macros to send any text, which is then subject to further expansion. These `when...()` macros are:

- `whenlatex(text)`: sends `text` when in LaTeX conversion mode,
- `whenhtml(text)`: sends `text` when in HTML conversion mode,
- `whenman(text)`: sends `text` when in man conversion mode,
- `whenms(text)`: sends `text` when in ms conversion mode,
- `whentxt(text)`: sends `text` when in ASCII conversion mode,
- `whensgml(text)`: sends `text` when in SGML conversion mode.

Once again, **note** that the difference between the `when`*format*`()` macros and the *format*`command()` macros is, that the former expands their argument while the latter does not. As an example, consider the following code fragment:

```
You are now reading
    whenlatex(a LaTeX-generated
              footnote(LaTeX is a great
               document language!)
              document)
    whenhtml(a HTML document via your
             favorite browser)
```

The `when`*format*`()` macros are used here to make sure that the arguments to the macros are further expanded; this makes sure that the `footnote` macro in the `whenlatex` block gets treated as a footnote.

4.3.2 Figures

Figures in format-independent documents are a problem. You *cannot* avoid contact with the final format (HTML, LaTeX or whatever) if you want to include figures in a text.

Yodl approaches figures as follows:

- Figures can only be included in LaTeX, HTML and XML documents.
- For LaTeX, you must prepare a picture in an external file that is included in the document as en *encapsulated PostScript* file. Incidentally, that means that `epsf` must be stated as one of the LaTeX styles using the `latexoptions` macro. The default, however, can be modified using the `setlatexfigureext()` macro.
  The file in question is stated in Yodl without an extension. Yodl provides a default extension, `.ps`.

- For HTML and XML, you must prepare a picture in an external file that is placed in the document using the <`img src=...`> tag. The file must have the default extension (`.jpg`) or the extension specified with the `sethtmlfigureext()` macro.
- All other output formats do not include pictures in the document, but typeset something like *insert figure .. here*.

The macro to include a figure is called, appropriately, `figure`. It takes three arguments:

- The first argument is the filename. This name may include directories, but may not include the filename extension. The reason for this is, that Yodl supplies the correct extension once the output format is known.
- The second argument is the figure title, or the caption. Yodl prefixes this caption with the text *Figure xx:*, where *xx* is a number.
- The last argument is a label, which Yodl defines as a placeholder for the figure number.

For example, you might draw a picture or scan a photo and put it in a `.jpg` file, for usage with HTML documents. The conversion to PostScript could be automated, e.g., using a Yodl macro:

```
SYSTEM(xpmtoppm picture.xpm | pnmtops > picture.ps)
```

See section 3.1.64 for details about using the `SYSTEM` macro.

After this, you would be reasonably safe that the picture is available for both HTML and LaTeX output. The picture would be typeset in a figure using:

```
figure(picture)
(A photo of me.)
(photo)
```

Note how the first argument, the filename, does not contain an extension. The third argument, which is a label, can be used in, e.g.,

```
See figure ref(photo) for a photograph showing me.
```

Yodl has a several auxiliary macros, which are:

- `fig(label)`: This macro is a shorthand for `getfigurestring() ref(label)`. It just makes typing shorter, and is used as e.g.: `See fig(photo) for a photograph.` Note that the string `figure` that is generated by this macro can be (re)defined, see below.

- `setfigurestring(name)`: This macro is similar to `setchapterstring` etc.. It defines the string that is used to identify a figure, and is (appropriately) `figure` by default. The macro `getfigurestring()` expands to the string in question. See also section 4.3.5 for a discussion of national language support.
- `sethtmlfigureext(.new)`: This macro redefines the filename extension for HTML conversions from `.gif` to `.new`. Note that you must include a leading dot in the redefinition.
  The new extension is used in the first following `figure` statement.
- `sethtmlfigurealign(align)`: This redefines the alignment of figures in HTML, which is default `bottom`. Check your HTML handbook for possible options; `top` and `center` should be fairly standard.
- `setlatexfigureext(.new)`: Redefines the extension from `.ps` to `.new`.

4.3.3 Fonts and sizes

Yodl's standard macro package supports the following macros to change fonts:

- `bf(text)`: sets `text` in boldface.
- `em(text)`: sets `text` emphasized, usually in italics.
- `tt(text)`: sets `text` in teletype.

Furthermore, the `tt()` macro does *not* expand macros occurring inside its argument. That means that you can safely write:

```
In Yodl, you can use tt(includefile(somefile)) to include a file
in your document.
```

The `tt()` macro should not be used for long listings of verbatim text; use `verb()` to set code samples etc..

Yodl's standard macro package has no commands to change font sizes, as the size is changed internally when appropriate (e.g., in section titles), nor is there a default macro to define other font-families.

4.3.4 Labels, links, references and URLs

References such as *see ... for more information* are very common in documents. Yodl supports three mechanisms to accomplish such references:

Labels and references: Labels can be defined in a document as a placeholder for the last number used in a sectioning command. At other points in the document, references to those labels are used. The reference expands to the number, as in *see section 1.3*.

This mechanism is available in all output formats. Furthermore, the numeric reference (1.3 in the example of the previous paragraph) is in HTML a clickable reference that leads to the mentioned section.

Labels and links: This mechanism can be used to set links in a document without using the number of a sectioning command, as in *see the introduction for more information*, with the *introduction* being a clickable link to some label.

This mechanism of course only leads to a clickable link in HTML: in other formats the text *see the* etc. is just typeset as is.

URLs: Universal Resource Locators (URLs) are used to create links to other HTML documents or services, like HTML's <a href=..> method. The URLs of course only result in clickable links in HTML output; in other output formats only some descriptive text appears.

The above mechanism is implemented by the following macros:

- The macro `label(name)` defines a label named `name`. The name of the label can be used in a `ref` or `link` macro.
- The macro `ref(name)` sets a reference to the label named `name`. The text of the reference is the number of the last sectioning command that was active during the creation of the label. When using references it is therefore important to define the corresponding labels right after a sectioning command, as in

```
section(How to install my program) label(howtoinstall)
This section describes...
...
See section ref(howtoinstall) for installation instructions.
```

  The macro `ref(howtoinstall)` expands to the number of the section named *How to install my program.*

- The macro `link(description)(name)` always expands to the `description`. In HTML output, a clickable link is created pointing to a label called `name`. For example:

```
label(megahard)
COMMENT(sigh...)
The Jodel package isn't shareware, it isn't
beggarware, it isn't freeware, it's
bf(megahard-ware).
...
Who wants a link(picosoft)(megahard)?
```

  This code fragment would always set the text *picosoft*, but under HTML a clickable link would appear pointing to `link(the text)(megahard)`.

- The macro `url(description)(location)` always expands to the `description`, but creates a hyperlink pointing to `location` in HTML. For example,

```
Take a look at my
url(homepage)(http://www.somewhere.nl/karel/karel.html).
```

The text `homepage`[1] always appears, but only in HTML it is a link. (Note that the double quotes, which are necessary in HTML around the location, are not required by Yodl.) To use a different font in the `description` part, surrond it *inside the url parameter list*, as in:

```
The Yodl package can be obtained at the site tt(ftp.rug.nl) in the
directory url(tt(/contrib/frank/software/yodl))
             (ftp://ftp.rug.nl/contrib/frank/software/yodl).
```

- The macro `email(address)` is a special case of `url`: under HTML, the `address` appears as a clickable link in slanted font to mail `address`. For example:

```
I can be reached at
email(f.b.brokken@rug.nl).
```

I can be reached at f.b.brokken@rug.nl<`f.b.brokken@rug.nl`>.

Always keep in mind that the name of a label must be exactly identical in both the `label` macro and in the `ref` or `link` macro. Other than that, the name is irrelevant.

Furthermore, note that `lincludefile` is yet another macro defining a label: it includes a file and automatically creates a label just before the included file's text. That means that a Yodl file like:

```
chapter(Introduction)
sect(Welcome)
lincludefile(WELCOME)(welcome)

chapter(Technical information)
lincludefile(TECHINFO)(techinfo)
```

creates two labels: `WELCOME` and `TECHINFO`.

Here are some final thoughts about using labels and references:

- Don't put 'weird' characters in label names. Generally, don't use spaces and tabs.
- The name of the label is always only an internal symbol; it does not appear in the output. Therefore, constructions such as the following are not correct:

```
ref(em(labelname))
```

[1] http://www.somewhere.nl/karel/karel.html

The reason for the incorrectness is, what internal name should `em(labelname)` generate? Here probably an attempt is made to set a reference in italics. The right construction is of course to set *whatever* ***`ref()`*** *returns* in italics, as in:

```
em(ref(labelname))
```

- The `label` macro should not appear nested inside another macro. There is no strict reason for this as far as Yodl is concerned; however, the processors of Yodl's output might go haywire. E.g., beware of the construction

  ```
  section(Introduction label(intro))
  ```

  The right form being

  ```
  section(Introduction)label(intro)
  ```

  (linking to `intro` usually does *not* show `Introduction`), or:

  ```
  label(intro)section(Introduction)
  ```

  (linking to `intro` usually *does* show `Introduction`).

4.3.5 Lists and environments

Yodl's default macros support the following lists and environments:

By default, the following lists are available:

Description lists: A description list consists of a list of elements, where each element starts with a short (usually bold faced) description. The description list is generated by the `description()` macro. The elements of the list start with `dit()`. The `dit()` macro expects a short description of the item.

Example:

```
A description list:
description(
dit(First this:) One item.
dit(Then this:) Another item.
)
```

Enumeraton lists: An enumeration list consist of sequentially numbered elements. The list is generated by the `enumeration()` macro. Its elements start with the `eit()` macro.

Example:

```
An enumerated list:
enumeration(
eit() One item.
eit() Another item.
)
```

Itemized lists: An itemized lists consists of indented items, usually preceded by a bullet.

An itemized list is produced by the `itemization()` macro, which has one argument: the items themselves. These items must start with `it()`.

Example:

```
An itemized list:
itemization(
it() One item.
it() Another item.
)
```

Specialized environments are:

Centered text: Centering text may not be available in all output formats. When unavailable, the text is typeset left-flushed.

Centered text is generated by the `center()` macro. Line brakes within centered text may be obtained using the `nl()` macro.

Example:

```
center(
Centered text. nl()
Another line of centered text.
)
```

Verbatim text: *Verbatim* text appears on the output exactly in the same layout as it is in the input file. Typesetting text in verbatim mode is useful for, e.g., source files. Depending on the output format, the font of the verbatim text is changed to a teletype font.

The text must either be inside the `verb()` macro. For example:

```
verb(
    This is totally verbatim text.
    It is not further processed by Yodl.
)
```

The verbatim text is of course not subject to macro expansion by Yodl. Note, however, that `SUBST` transformations *will* take place, as these substitutions take place during the lexical scanning phase of `Yodl`'s input, and are not part of the macro-expansion process. See also section 3.1.62.

Furthermore, if a character translation table has been defined, the argument of the `verb()` macro will also be subject to character table transformations. By temporarily suppressing the active character table (see section PUSHCHARTABLE 3.1.53) this can be prevented.

Quotations: Quotations are usually indented with respect to their surrounding text. It is for the author to decided whether the quoted text should be typeset normally, or that it should be bold-faced or emphasized. To insert a quotation use the `quote()` macro:

```
Shakespeare once wrote:
quote(
    ``To be or not to be, that's the question''
)
```

National language support

Yodl includes rudimentary national language support (NLS), in the sense that it allows you to redefine the strings identifying chapters or parts, or the strings identifying figures. E.g., a command `chapter(Introduction)` by default results in the text *Chapter 1: Introduction*.

Using the `setchapterstring(text)` macro, the *Chapter* text can be redefined. E.g., in a Dutch text you might put

```
setchapterstring(Hoofdstuk)
```

somewhere near the beginning of your document. Similar to `setchapterstring`, a macro `getchapterstring` exists returning the text identifying chapters. (Internally, `getchapterstring` is of course used to actually set the text). To redefine the text to identify a part, use `setpartstring(text)`; to redefine the text to identify a figure, use `setfigurestring(text)`.

The `set....string` macros only influence how Yodl names chapters or parts in HTML, `man`, `ms` or `txt` output. LaTeX output is not affected, since LaTeX does its own NLS. Usually, NLS is present for LaTeX as a 'style file' named, e.g., `dutch.sty`. Therefore, if you want a Dutch document, you need to:

- put `latexpackage(dutch)(babel)`in the preamble of the document. This ensures that LaTeX uses Dutch abbreviation rules.
- redefine the chapter and part names for non-LaTeX output, using:

```
setlanguage(dutch)
```

- Finally, you should probably type your text in Dutch.

The `setlanguage()` macro expects one argument: the name of the language that is used. See section 4.2 for details about this macro. The `setlanguage()` macro redefines the language-dependent section (and other) headers, and depends on the availability of the corresponding `language<name>()` macro, where `<name>` is the name of the language (by convention `<name>` states the english name of the language). Currently, `languagedutch()`, `languageenglish()` (the default), and `languageportugese()` are available. It's easy to expand this little set with macros for other languages. The `setlanguage()` macro merely requires the specification of the language. For example:

```
setlanguage(english)
```

This macro installs the following defaults (corresponding translations should be defined for other languages):

```
settocstring(Table of Contents)
setaffilstring(Affiliation)
setauthorstring(Author)
setchapterstring(Chapter)
setdatestring(Date)
setfigurestring(Figure)
setpartstring(Part)
settitlestring(Title)
```

Pagebreaks after the title and table of contents

Yodl inserts page-breaks in a limited number of cases:

- A pagebreak is generated after the title information in `book` and `report` documents.
- A pagebreak is generated after a table of contents in all documents.

So, when a document has both title information and a table of contents then whatever follows next normally starts on a separate page. Furthermore, if the document is a `book` or a `report`, the title and table of contents will also be separated by pagebreaks.

This behavior can be modified using the `(no)titleclearpage()` and `(no)tocclearpage()` directives, further described in section 4.3.7.

4.3.6 Sectioning

This section describes the sectioning commands for `article`s, `report`s, `book`s and for `plainhtml`. The document type `manpage` defines its own sectioning commands (cf. section 4.1.2:

- `part(title)`: Starts a new part. Only available in `book` documents.
- `chapter(title)`: Starts a new chapter. Only available in `book` or `report` documents.
- `sect(title)`: Starts a section.
- `subsect(title)`: A subsection.
- `subsubsect(title)`: A sub-subsection.
- `subsubsubsect(title)`: An even smaller sectioning command.

These macros generate entries in the table of contents and use numbering, which means that each section is prefixed with a number (1, 1.1, 1.2, and so on). The macros are also available with an `n` prefix (`npart`, `nchapter`, `nsect` etc.) which generate neither entries in the table of contents nor numbers. The `n`-versions can be used in, e.g., an article where the sectioning commands should show their captions, but not any numbers generated by default.

Sectioning should always start at the top level sections of the available document: `chapter` for reports, `sect` for articles, etc.. If you start a document with a lower sectioning command (e.g., when you start an article with a `subsect`), the numbering of sections may go haywire. The only exception to this rule is the `part` of a `book` document: parts are optional, in books, `chapter`s may be the top sectioning commands. Summarizing, books or reports should start with `chapter`. Articles should start with `sections`.

The sectioning commands have a further function: when `label` statements appear after the sectioning command, then a label name is used as a placeholder for the last generated number. This is further described in section 4.3.4.

4.3.7 Typesetting modifiers

This section lists various macros that can be used to modify the looks of your document. When used, these macros must appear *before* stating the document type with `article`, `report`, `book`, `manpage` or `plainhtml`.

- `abstract(text)`: This macro is relevant for all output formats. The `text` is added to the document after the title, author and date information, but before the table of contents. The abstract is usually set as a quote, in italics font (though this depends on the output format). Abstracts are supported in `article`s and `report`s, but not in other document types. I.e., if you need introductory text in a `book`, you should start with a non-numbered chapter that holds this text.
- `affiliation(site)`: This macro is relevant for `article`, `report` and `book` documents. It defines the affiliation of the author. The `site` information appears in the title, below the author's name.
- `htmlbodyopt(option)(value)`: This macro adds `option="value"` to the <`body`> tag that is generated for HTML output. The HTML converter generates <`body`> tags each time that a new file is started; i.e., at the top of

the document and at each chapter-file. Different HTML browsers support different <body> tag options, but useful ones may be e.g.:

```
htmlbodyopt(fgcolor)(#000000)
htmlbodyopt(bgcolor)(#FFFFFF)
```

This defines the foreground color as pure white (red/green/blue all 0) and the background color as black (red/green/blue all hexadecimal FF, or 255). Another useful option may be `htmlbodyopt(background) (some.gif)`, defining `some.gif` as the page background.

See the documentation on HTML for more information.

Note that `value` is automatically surrounded by double quotes when this macro is used. They should not be used by authors using this macro.

- `latexdocumentclass(class)`: This macro forces the `\documentclass{...}` setting in LaTeX output to `class`.
- `latexlayoutcmds(commands)`: This macro can be used to specify your own LaTeX layout commands. When present, the `commands` are placed in LaTeX output following the `\documentclass` definition.
- `latexoptions(options)`: This macro is only relevant for LaTeX output formats, it is not expanded in other formats. The `options` are used in LaTeX's `\documentclass` definition; e.g., a useful option might be `dina4`. Multiple options should be separate by commas, according to the LaTeX convention.
- `latexpackage(options)(name)`: This macro is only relevant for LaTeX output formats, it is not expanded in other formats. Each *package* should have its own `latexpackage()` statement. If there are no options, the `options` argument should remain empty. Here is an example using this macro:

  ```
  latexpackage(dutch)(babel)
  ```

- `mailto(email)`: The `mailto` macro is only expanded in HTML documents, it is ignored in other formats. It defines where mail about the document should be sent to.
- `nosloppyhfuzz()`: By default, the LaTeX output contains the text

  ```
  \hfuzz=4pt
  ```

  which is placed there by the macro package. This suppresses *overfull hbox* warnings of LaTeX when the overfull-ness is less than 4pt. Use `nosloppyhfuzz()` to get the standard LaTeX warnings about overfull hboxes.
- `notableofcontents()`: As the name suggests, this macro suppresses the generation of the table of contents. For HTML that means that no clickable index of sections appears after the document title.

  The table of contents is by default suppressed in `plainhtml` and `manpage` documents.

- `notitleclearpage()`: Normally, Yodl inserts a `clearpage()` directive after typesetting title information in `book` or `report` documents, but not in `article` documents. Use `notitleclearpage` to suppress this directive.
- `notocclearpage()` (no table-of-contents clear-page): In all document types, Yodl inserts a `clearpage()` directive following the table of contents. Use `notocclearpage()` to suppress that.
- `noxlatin()`: The LaTeX output contains by default the command to include the file `xlatin1.tex`, distributed with Yodl. This file maps Latin-1 characters to LaTeX-understandable codes and makes sure that you can type characters such as ü, and still make them processable by LaTeX. If you don't want this, put `noxlatin()` in the preamble.
- `standardlayout()`: This is another LaTeX option. Use `standardlayout()` to get 'vanilla' LaTeX layout, possibly indenting paragraphs and using fairly limited vertical spacing between paragraphs. This macro is ignored for other conversion types.
- `titleclearpage()`: Forces the insertion of a `clearpage()` directive after the title information has been typeset. This behavior is the default in `book` and `report` documents. See also `notitleclearpage()`.
- `tocclearpage()`: Forces the insertion of a `clearpage()` directive following the table of contents. This behavior is default in all document types; the macro is provided for consistency reasons with `(no)titleclearpage()`.

Note again: these modifiers must appear *before* the document type definition.

4.3.8 Miscellaneous commands

The following is a list of commands that don't fall in one of the above categories.

- `clearpage()`: This macro starts a new page in LaTeX. For HTML, a horizontal rule is shown. (Note that the macro package sometimes inserts new pages by itself; e.g., following a table of contents. See also section 4.3.7 for a discussion of `(no)titleclearpage()` and `(no)tocclearpage()`.)
- `def(macro)(nrofarguments)(definition)`: This defines a new macro `macro` having `nrofarguments` arguments, and expanding to `definition`. The markers `ARG`x, where x is 1, 2, etc., can be used in the `definition` part to indicate where arguments should be pasted in. This macro is a shorthand for `DEFINEMACRO`, see section 3.1.11.
- `footnote(text)`: This macro sets `text` as a footnote when the output format allows it. When not, the text is set in parentheses.
- `gagmacrowarning(name name ...)`: This macro suppresses `yodl`'s warnings *cannot expand possible user macro name*, where `name` is a candidate macro name. `gagmacrowarning` is a synonym for `NOUSERMACRO`, described in section 3.1.41.
  E.g., if your document contains `"as for manpages, see sed(1), tr(1) and awk(1)"`, and if you get tired of warnings about possible user macros sed, tr and awk, try the following:

```
gagmacrowarning(sed tr awk)
...
As for manpages, see sed(1), tr(1) and awk(1).
```

- `htmlnewfile()`: Starts a new subfile in HTML output. This stanza is also automatically generated when the HTML converter encounters a `chapter` directive. Using `htmlnewfile`, the output can be split at any point. However make sure that the subfile is still reachable; e.g., by creating a clickable link with `label` and `ref`, or `label` and `link`.

- `includefile(file)`: Includes `file` and defines a label (see the `label` macro) with the same name. Furthermore, a message about the inclusion is shown on the screen. The `file` is searched for relative to the directory of the file in which the `includefile` macro was used (or relative to the directory where the `yodl` run was started when the `+NOTRANS(-{}-{})legacy-include` or `-L` option was provided) and also in the system-wide include directory. The default extension `.yo` is supplied if necessary.
  The `lincludefile` macro is handy in the following situation:

```
chapter(Introduction)
lincludefile(INTRO)(intro)
```

  This fragment starts a chapter and includes a file. Here the label name (`INTRO`) can also be used to refer to the chapter as the `lincludefile` stanza appears *immediately* following the corresponding sectioning command.

- `nl()`: Forces a new line. Some output formats may produce an error upon the usage of `nl()` in 'unexpected' places; e.g., LaTeX won't allow new lines in the footnote text (as defined in the `footnote` macro). Using `nl()` in running text should however be ok. Example:

```
This line is nl()
broken in two.
```

- `redefinemacro(macro)(nrofargs)(redef)`: This command (re)defines a macro, expecting `nrofargs` arguments, to `redef`. If a previous definition of the macro existed, it is overruled. Example:

```
redefinemacro(clearpage)(0)(\
em(---New page starts here---))
```

  Use `ARG`x in the `redef` part to indicate where all arguments should occur, as in the following imaginary macro to typeset a literature reference:

```
redefinemacro(litref)(3)(
    Title: bf(ARG1) nl()
    Author(s): em(ARG2) nl()
```

```
    Published by: ARG3
)
...
litref(Java in a Nutshell)
  (David Flanagan)
  (O'Reilly & Associates, Inc.)
```

The `redefinemacro` statement also has a shorthand called `redef`.

4.4 Locations of the macros

The files defining the macros are by default installed to the directory `/usr/local/share/yodl` during Yodl's installation process (Note that this diverts from an earlier default: `/usr/local/lib/yodl`; furthermore, some systems or some distributions may use other locations).

The files in this directory are organized as follows:

- The files that should be read for a particular conversion are named after their conversion, e.g., `latex.yo` and `html.yo`. These files must be processed by Yodl before your document can be converted accordingly. The provided `yodl2...` scripts take care of that automatically.
- All support counters, symbols and macros are defined in files named `std.<conversion>.yo`, e.g., `std.html.yo, std.latex.yo`. These files may be modified without notice, and are an essential part of the Yodl macros. They should not be modified by hand, as they are created by the macro generating process.
- The predefined character tables are found in files names `chartables/<conversion>.yo`.

The (binary) Yodl package contains the following programs and support files:

- The `yodl` program itself, which generates converted document(s);
- The `yodlpost` postprocessor, which performs fixups for conversion formats. Using `yodlpost` is required for formats whose documents cannot be created in one pass by `yodl` itself;
- Auxiliary scripts such as `yodl2tex`, `yodl2html`;
- The macros and character tables for the various conversion types;
- The raw macros and the macro-generating scripts;
- The documentation (html and manual pages)

The source Yodl package contains all the sources files, installation guides, changelogs etc., that are required to compile the binary programs. Those who want to compile Yodl themselves, must have a `C` compiler (preferably the `Gnu` **C** compiler) available, and preferably the `icmake` program maintenance utility. Basic support for `make` is provided as well.

Chapter 5

Conversions and converters

Each macro package handling a conversion from Yodl to a given output format has its peculiarities. Although the various macro packages are very similar, they do show some differences, due to the unique characteristics of the output formats. Normally, these differences should not cause difficulties in performing the conversion(s). In this chapter the conversion of a Yodl document is covered. The currently supported document types are discussed. Furthermore, in this chapter the new *post processor* `yodlpost` is described as well as a little support program: `yodlverbinsert`.

5.1 Conversion script invocations

Yodl is distributed with scripts named `yodl2latex`, `yodl2html` and other `yodl2...` drivers. Invocations like

```
yodl2latex file
```

causes `Yodl` to process `file.yo` and to write output to `file.latex`. The extension of the input file, `.yo`, is the default `Yodl` extension; the extension of the output file, `.latex`, is given by the name of the shell script. Analogously, `yodl2html` writes to a file having the extension `.html`.

The conversion scripts auto-load the macro file appropriate for the conversion: `latex.yo` for LaTeX conversions, `html.yo` for HTML conversions, etc.. The macro files are in `Yodl`'s standard include directory (which is mentioned in `Yodl`'s usage information when `Yodl` is started without arguments). If the include directory is altered in such a way that it doesn't contain a path to the default directory anymore, then `Yodl` won't be able to auto-load the conversion specific macro files, producing unexpected results. This can be prevented by specifying the literal text `$STD_INCLUDE` in a user-defined path setting.

When the conversion scripts themselves are started without arguments, usage information is shown about the conversion scripts.

Depending on the conversion type, the following output is produced:

- For LaTeX conversions, one output file with the extension `.latex` is written.
- For HTML conversions, several files may be written; one file per chapter of the original document. When the document is not sectioned by chapters, only one output file is produced.

  The 'main' output file always has the name of the input file but with extension `.html`. This file holds the document title and the table of contents. When more than one output files are created, then they are named `name01.html`, `name02.html` etc., where `name` is the original name of the input file. E.g., a document `prog.yo` might lead to `prog.html`, `prog01.html` etc..
- For man conversions, one output file with the extension `.man` is written.
- For text conversions, the converter is named `yodl2txt` and one output file with the extension `.txt` is created.
- For XML conversions, the converter is named `yodl2xml` and output files are produced comparably to the way they are produced with the `html` conversion: one file per chapter if chapters are used, otherwise one single output file, having the extension(s) `.xml`.

The 'second-phase' scripts, distributed with earlier versions of `Yodl`, are no longer part of `Yodl`'s distribution, as they do not relate directly to `Yodl`'s actions. They may remain useful, though, as leftovers from earlier distributions.

5.2 The HTML converter

HTML doesn't support automatic section numbering or resolving of label/reference pairs. The converter takes care of this. Other target languages (e.g., XML, text) suffer from the same problems.

Direct commands to HTML

Similar to the LaTeX converter, you can use either `NOTRANS` or `htmlcommand` to send HTML commands to the output. Or, since the only 'difficult' characters are probably only < and >, you can also resort to `CHAR` for these two characters.

Furthermore, the HTML converter defines the macro `htmltag`, expecting two arguments: the tag to set, and an 'on/off' switch. E.g., `htmltag(b)(1)` sets while `htmltag(b)(0)` sets .

E.g., the following code sends a HTML command <hr> to the output file when in HTML mode:

```
COMMENT(-- alternative 1, using htmlcommand --)
    htmlcommand(<hr>)

    COMMENT(-- alternative 2, using NOTRANS --)
    IFDEF(html)(
        NOTRANS(<hr>)
    )()
```

```
COMMENT(-- alternative 3, using CHAR --)
IFDEF(html)(
    CHAR(<)hrCHAR(>)
)()

COMMENT(-- alternative 4, using htmltag --)
htmltag(hr)(1)
```

Section numbering

The HTML converter numbers its own sections. This is handled internally. However, the current converter only can number sections as starting at 1, and outputs the numbers in arabic numerals (you can't number with A, B, etc..).

5.3 The LaTeX converter

The LaTeX converter is, from `Yodl`'s viewpoint, an easy one: since LaTeX supports wide functionality, a Yodl document is basically just re-mapped to LaTeX commands. No post-processing by `yodlpost` is required.

Direct commands to LaTeX

To send LaTeX commands directly to the output, use the `latexcommand()` macro (see section 4.3.1), or use `NOTRANS` (see section 3.1.40). The advantage of the `latexcommand` macro is that it only outputs its argument when in LaTeX mode.

The following two code fragments both output `\pagestyle{plain}` when in LaTeX mode:

```
COMMENT(-- First alternative: --)
latexcommand(\pagestyle{plain})

COMMENT(-- Second alternative: --)
IFDEF(latex)(
    NOTRANS(\pagestyle{plain})
)()
```

Verbatim text

The Yodl macro package defines two macros that generate verbatim text (e.g., source code listings). These macros are `verb()` and `tt()`.

verb The `verb()` macro and is meant for longer listings (whole files); as in:

```
    verb(
#include <stdio.h>

int main (int argc, char **argv)
{
    printf ("Hello World!\n");
    return 0;
}
    )
```

The `verb()` macro generates `\begin{verbatim}` and `\end{verbatim}` when used in LaTeX conversion mode. That means that (in that situation) the `verb` macro has only one caveat: you cannot put `\end{verbatim}` into it.

tt The `tt()` macro also inserts verbatim text. It is used for short in-line strings (e.g, `**argv`). The LaTeX converter doesn't actually use a verbatim mode, but sets the characters in teletype font.

5.4 The man converter

Manual pages can be constructed using the special `yodl2man` converter. This converter assumes that the manual page has been designed using the `manpage()` macro. Yodl (and thus the `yodl2man` converter, when converting man-pages, skips all leading white space on lines. Paragraphs are supported, though. An empty line separates paragraphs.

Direct commands to man

Either `NOTRANS` or `mancommand` can be used to send man commands to the output.

E.g., the following code sends a MAN command <`hr`> to the output file when in MAN mode:

```
COMMENT(-- alternative 1, using mancommand --)
    mancommand(<hr>)

    COMMENT(-- alternative 2, using NOTRANS --)
    IFDEF(man)(
        NOTRANS(<hr>)
    )()
```

5.5 The txt converter

Plain text documents can be constructed using the `yodl2txt` converter. This converter resolves all references into the document itself, so postprocessing is required.

Direct commands to txt

Either `NOTRANS` or `txtcommand` can be used to send txt commands to the output.

E.g., the following code sends a TXT command <hr> to the output file when in TXT mode:

```
COMMENT(-- alternative 1, using txtcommand --)
    txtcommand(<hr>)

    COMMENT(-- alternative 2, using NOTRANS --)
    IFDEF(txt)(
        NOTRANS(<hr>)
    )()
```

5.6 The experimental XML converter

The XML converter is experimental. It was added to `Yodl` to allow me to write documents for the horrible 'webplatform' of the university of Groningen. The XML support files (located in the `xml` directory in the standard macro's directory) clearly reflect this target. Although experimental, they were kept because the XML macros support interesting constructions allowing `Yodl` to handle closing tags somewhat more strict than required for HTML.

5.7 The Yodl Post-processor 'yodlpost'

Following the conversion of a `Yodl` text, most target-languages require an additional operation, called 'post-processing'. Post-processing is required for various reasons: to split the output in separate files (HTML, XML); to fixup the locations of labels, that are referred to earlier than the labels are defined (virtually all target language except LaTeX); tables of contents are available only after the conversion, but have to be inserted at the beginning of the document; etc. etc..

Starting with `Yodl` V. 2.00 there is only one post-processor, handling all the conversions for all target languages. Program maintenance of just one program is certainly easier than maintenance of as many programs as there are target-languages, at the expense of only a slightly larger program: after all, the one post-processor contains the conversion procedures for all target languages. It turns out that this is a very minimal drawback. See section 6.7 for the technical details of post-processor program maintenance.

The post-processor that is distributed since `Yodl` V. 2.00 does *not* use the `.tt(Yodl)TAGSTART.` and `.tt(Yodl)TAGEND.` tags anymore. Instead, the conversion process produces a *index file* in which comparable information is written. The advantage of using an index file is that the postprocessor doesn't have to parse the output file generated by `Yodl` twice (once to determine the tags, once to process the tags), which by itself accelerates the conversion process; and (albeit of a somewhat limited practical importance) that the tags are no longer *reserved words*: authors may put `.tt(Yodl)TAGSTART.` and `.tt(Yodl)TAGEND.` into their texts as often as they want.

Authors should be aware of some caveats with respect to some target languages:

man- and ms- conversions all dots are converted by the active character conversion table to `\&.`. Commands in these languages always start with a dot as the first character on a line. In order to insert these commands the `roffcmd()` (see section `MACROLIST`) should be used.

plain text conversions As stated before, the ASCII converter basically only strips macronames from its input. This converter is so basic, that it should only be used as a last resort, when no other target language is available for the job. With the plain text converer, the layout of the input file is very important, as the output is basically the same as the input. The only exception to this rule are multiple empty lines, which normally are consumed by the post-processor, to be replaced by one single empty line.

sgml conversions the SGML converter was implemented for historic reasons. It is by no means complete, and can at best be considered an 'initial starting point'. Currently, the SGML converter only supports the `article` document type, having `sect` as its top-level sectioning command.

xml conversions The XML converter was implemented to allow me (Frank) to produce XML text as defined by the so-called 'webplatform' of the University of Groningen. A completely pathological implementation of XML, crippling its users to the level of the 'double click brigade'. Well, so be it. The net result of this is that `Yodl` now offers some basic form of XML conversion. After adding XML conversion, however, there was never any real situation where it could profitably be used. Therefore, further development of the XML conversion has, for the time being, been suspended.

5.8 The support program 'yodlverbinsert'

The program `yodlverbinsert` is a simple **C** support program that can be used to generate `verb()`-sections in `Yodl` files from sections of existing files. The files from which sections are included are usually **C** or **Cpp** source files, accepting either `//` or `/*`-style comment.

`Yodlverbinsert` offers the possibility to indent both the initial `verb`-statement and the inserted file contents. Furthermore, an additional empty line may be inserted before the first line that is actually inserted. The program is invoked according to the following synopsis:

yodlverbinsert [OPTIONS] `marker file`

The arguments have the following meanings;

- `marker`
  The argument `marker` must start in `file`'s first column en must either start as a standard **C** or **C++** comment: `//` or `/*` must be used. Following that, the remainder of the argument is used as a label, e.g., `//label`, `/*LABEL*/`. The label may contain non-alpha characters as well. Except for the first two characters and their locations no special restrictions imposed upon the label

texts. A labeled section ends at the next `//=` (when the label started with `//`) or at the next `/**/` (when the label started with `/*`). Like the labels, the end-markers must also start in the file's first column.

- `file`
  The argument `file` must be an existing file. `Yodlverbinsert` was designed with **C** or **C++** sources in minde, from which labeled sections must be inserted into a `Yodl` document, but `file` could also refer to another type of (text) file.

The default values of options are listed below, with each of the options between square brackets. The defaults were chosen so that `yodlverbinsert` performs the behavior of an earlier version of this program, which was not distributed with `Yodl`.

- **-N**
  Do not write a newline immediately following `verb`-statement's open-parenthesis. By default it is written, causing an additional line to be inserted before the first line that's actually inserted from a file.
- **-s** `spaces` [0]
  start each line that is written into the `verb`-section with `spaces` additional blanks.
- **-S** `spaces` [8]
  prefix the `verb` of the `verb`-section by `spaces` additional blanks.
- **-t** `tabs` [0]
  start each line that is written into the `verb`-section with `tabs` additional tab characters. If both `-s` and `-t` are specified, the tabs are inserted first.
- **-T** `tabs` [0]
  prefix the `verb` of the `verb`-section by `tabs` additional tab characters. If both `-S` and `-T` are specified, the tabs are inserted first.

`Yodlverbinsert` writes its selected section to its standard output stream.

5.8.1 Example

Assume the file `demo` contains the following text:

```
preceding text

//one
one 1

//=

/*two*/

    two

/**/
```

```
trailing text
```

Then the following commands write the shown output to the program's standard output:

- `verbinclude //one demo`

```
        verb(
one 1

)
```

- `verbinclude -N //one demo`

```
        verb(one 1

)
```

- `verbinclude -s4 '/*two*/' demo`

```
        verb(

        two

)
```

To call `yodlverbinsert` from a `Yodl` document, use **PIPETHROUGH**. E.g.,

```
PIPETHROUGH(yodlverbinsert //one demo)
```

Alternatively, define a simple macro like the macro `verbinsert`:

```
DEFINEMACRO(verbinsert)(2)(PIPETHROUGH(yodlverbinsert //ARG1 ARG2)()\
)
```

which may be a useful macro if all or most of your labeled sections start with `//`, and if `yodlverbinsert`'s arguments don't vary much. Variants to this macro can easily be conceived of.

Chapter 6

Technical information

This chapter consists of various sections. The first section describes `Yodl` from the point of view of the systems administrator. Issues such as the installation of the package are addressed here. The second section describes `Yodl`'s technical implementation in some detail. Apart from the documentation about `Yodl` given here, much can be found in the individual source files. However, section 6.2 describes 'the broad picture'. Having read section 6.2, it should be relatively easy to determine what happens where inside the `Yodl` program and the `yodl-post` post processor.

6.1 Obtaining Yodl

`Yodl` and the distributed macro package can be obtained from `https://fbb-git.gitlab.io/yodl/`. The associated git-repository page is located at `https://gitlab.com/fbb-git/yodl`.

Yodl can also be retrieved as a binary Debian package: to find it at `http://www.debian.org/distrib/packages` by entering `yodl` (in any distribution) in the form-fields you'll find on that page.

6.1.1 Installing Yodl

The binary package, distributed in `yodl-X.Y.Z_a.b.c.deb` can be installed using `dpkg -install yodl-X.Y.Z`. It installs:

- `Yodl`'s binaries in `/usr/bin`;
- `Yodl`'s macros in `/usr/share/yodl`
- `Yodl`'s documentation in `/usr/share/doc/packages/yodl`;
- `Yodl`'s manpages in `/usr/share/man/man{1,7}`;

Local installations, not using the Debian installation process, can be obtained using the provided `icmake` build-script see below. An alternative (cf. `contrib/build.pl`) is to use the contributed `build.pl` script. Note that `build.pl` is not maintained by me. If you bump against problems when using it, I'll probably not be able to help you out.

If a local installation is preferred or required, unpack the file `yodl-X.Y.Z.tar.gz`. Next, chdir to the directory `yodl-X.Y.Z`, and optionally tweak the file `config` to your needs. Next, issue the command:

```
build programs    (or use 'build programs strip' to strip the binaries)
build macros
build man
build manual
```

followed by

```
install programs WHERE        (install the programs under WHERE,
                               e.g. install programs /usr/bin)

install macros WHERE          (install the macros under WHERE
                               e.g., install macros /usr/share/yodl)

install man WHERE             (install the man-pages under WHERE
                               e.g., install man /usr/share/man)

install manual WHERE          (install the manual under WHERE
                               e.g., install manual /usr/share/doc/yodl)

install docs WHERE            (install additional docs under WHERE
                               e.g., install docs /usr/share/doc/yodl)
```

The installation process installs the binaries, manual pages, other documentation and macro files under the indicated directories.

However, by far the easiest way to install a binary distribution is to use the Debian `dpkg +NOTRANS(-{}-{})install yodl*.deb` command. `Dpkg` installs the various parts according to Debian's conventions under `usr/`.

Installation from source requires you to have the following programs installed on your system:

- A recent **C** compiler and run-time environment. A POSIX-compliant compiler, libraries and set of header files should work without problems.
- A recent installation of `Icmake`: `Icmake` is part of the standard Debian distribution, and can also be obtained from `https://fbb-git.gitlab.io/icmake/`
- Standard tools, like `sed`, `grep`, `perl`, etc..
- `/bin/sh`: a POSIX-compliant shell interpreter. The GNU shell interpreter `bash` can be used instead.

6.2 Organization of the software

This section describes the organization of the source files. Its contents are not necessarily relevant for the binary distribution. The section is probably most useful to those readers who want to be able to extend or who want to do maintenance on `Yodl`'s sources, or who want simply to understand what's happening inside the `Yodl` program.

Much of the documentation is provided in the individual source files themselves. This section, however, should offer the 'broad picture', allowing you to understand the logic behind `Yodl` relatively fast.

6.2.1 Subdirectories and their meanings

After unpacking `Yodl`'s source archive, the following directories are available:

- `yodl`: the root-directory of the `Yodl` tree. All sources and program maintenance scripts are found in or below this directory.
- `yodl/macros`: This directory contains all the macro definitions of the standard macro package. It contains the following subdirectories:
  - `yodl/macros/in`: This directory contains generic macro files. These macro files contain the word `@STD_INCLUDE@`, which is replaced by the standard include directory used in a particular distribution.
  - `yodl/macros/rawmacros`: This directory contains the raw macro definition files themselves. One file per raw macro. A raw macro contains the implementations of that macro for *all* supported conversion types, and has the extension `.raw`. Furthermore, this directory contains some support scripts: `create, separator.pl, startdoc.pl`.
  - `yodl/macros/yodl`: this is the directory to contain `Yodl`'s standard macros. The (recursive) contents of this directory is eventually copied by the installation procedure to the `.../share/yodl` directory, which then becomes `Yodl`'s standard include directory.
  - `yodl/macros/yodl/chartables`: This directory contains character-translation tables for various target languages.
  - `yodl/macros/yodl/xml`: This directory contains the XML frame files, used to convert `Yodl` documents to XML, as implemented by the 'webplatform' of the University of Groningen. All these frame files have the extensions `.xml`.
- `yodl/man`: The raw source files of all man-pages: manpages of the `Yodl` program itself, of the yodl post-processor, of the conversion scripts, of the builtinfunctions, of the standard macros and of `Yodl`'s `manpage` and `letter` document types. These raw source files have the extensions `.in`, indicating that they may contain `@STD_INCLUDE@` words, which is replaced by the eventually used standard include path.
  - `yodl/man/1`: The destination for `Yodl`'s manual pages in section 1 (programs).
  - `yodl/man/7`: The destination for `Yodl`'s manual pages in section 7 (macro packages and conventions).

- `yodl/manual`: The source files of the complete `Yodl` manual, as well as the directories for the various converted formats. The script `build`, found in this directory, constructs the manual in the subdirectories:
  - `yodl/manual/html`: the HTML-converted manual;
  - `yodl/manual/latex`: the LaTeX-version of the manual;
  - `yodl/manual/pdf`: the pdf-version of the manual;
  - `yodl/manual/ps`: the PostScript-version of the manual;
  - `yodl/manual/txt`: the plain text-version of the manual;
- `yodl/manual/`: The source files of the complete The `Yodl` document files themselves are located in subdirectories of this directory. They are:
  - `yodl/manual/converters`
  - `yodl/manual/intro`
  - `yodl/manual/macros`
  - `yodl/manual/technical`
  - `yodl/manual/userguide` (and various subdirectories)
- `yodl/scripts`: support scripts used by the building process: `configreplacements` replaces `@XXX@` words by their actual values as found in `yodl/src/config.h`; `yodl2whatever.in` is the generic yodl-converter, calling macros specific for a particular conversion type. This generic converter is installed in `.../bin/`, together with specific converters, installed as soft-links to this generic converter.
- `Yodl`'s base directory contains sub-directories holding the source files of the **C** programs `Yodl` and `yodl-post`, as well as all auxiliary directories containing sources of the (logical) components of these programs. Most of these components are like **C++** classes in that they define a building block of the `Yodl` and/or `yodl-post` program. Their organization, interaction and relationship is described below. They are:
  - `yodl/args`: the component handling the command-line arguments;
  - `yodl/builtin`: the component handling `Yodl`'s builtin functions;
  - `yodl/chartab`: the component handling `Yodl`'s character table type;
  - `yodl/counter`: the component handling `Yodl`'s counter type;
  - `yodl/file`: the component handling all file operations (locating, opening, etc.);
  - `yodl/hashitem`: key/value combinations stored in `Yodl`'s hashtable;
  - `yodl/hashmap`: `Yodl`'s hashtable;
  - `yodl/lexer`: `Yodl`'s lexical scanner: this component consumes the `.yo` file, and produces a continuous stream of tokens to be handled by another component: the parser.
  - `yodl/lines`: the component storing lines of text, used by `yodl-post`.
  - `yodl/macro`: the component handling `Yodl`'s macro type;
  - `yodl/message`: the component handling all messages (warnings, errors, verbosity settings, etc.).
  - `yodl/new`: the component handling all memory allocations (except for duplicating *strings*, which is handled by the root-component).

- `yodl/ostream`: the component handling all `Yodl`'s output to its output-file (`Yodl` may also output to strings, which is not handled by the ostream component).
- `yodl/parser`: the component handling the tokens produced by the lexer-component. This component governs all actions to be taken during a conversion. Its actions all derive from its function `parser_process()`.
- `yodl/postqueue`: the component handling the postprocessing required by most conversions.
- `yodl/process`: the component handling the execution of child- or system-processes.
- `yodl/queue`: the component allowing the lexical scanner to queue its input, awaiting further processing.
- `yodl/root`: the component defining some basic typedefs and enumerations, as well as the `new_str()` function duplicating a string, and the `out_of_memory()` function handling memory allocation failures.
- `yodl/stack`: the component implementing a stack data structure.
- `yodl/string`: the component implementing a text-storage data structure and its functionality.
- `yodl/strvector`: the component implementing a table of `SUBST` key-value pairs.
- `yodl/subst`: the component handling `Yodl`'s SUBST definitions;
- `yodl/symbol`: the component handling `Yodl`'s symbol type;
- `yodl/yodl`: the sources of the `Yodl` program itself. This directory also contains the implementations of all builtin functions, whose filenames all start with `gram_` (E.g., `gramaddtocounter.c`).
- `yodl/yodlpost`: the sources of the `yodl-post` program.

The script `build`, found in this directory, compiles the programs and macros, storing them in:

- `tmp/install/usr/bin` (the binary programs)
- `tmp/install/usr/share/yodl` (the macros)
- `tmp/install/usr/share/man` (the man-pages)
- `tmp/install/usr/share/doc` (the manual and additional documentation)

6.3 Yodl's component interrelations and component setup

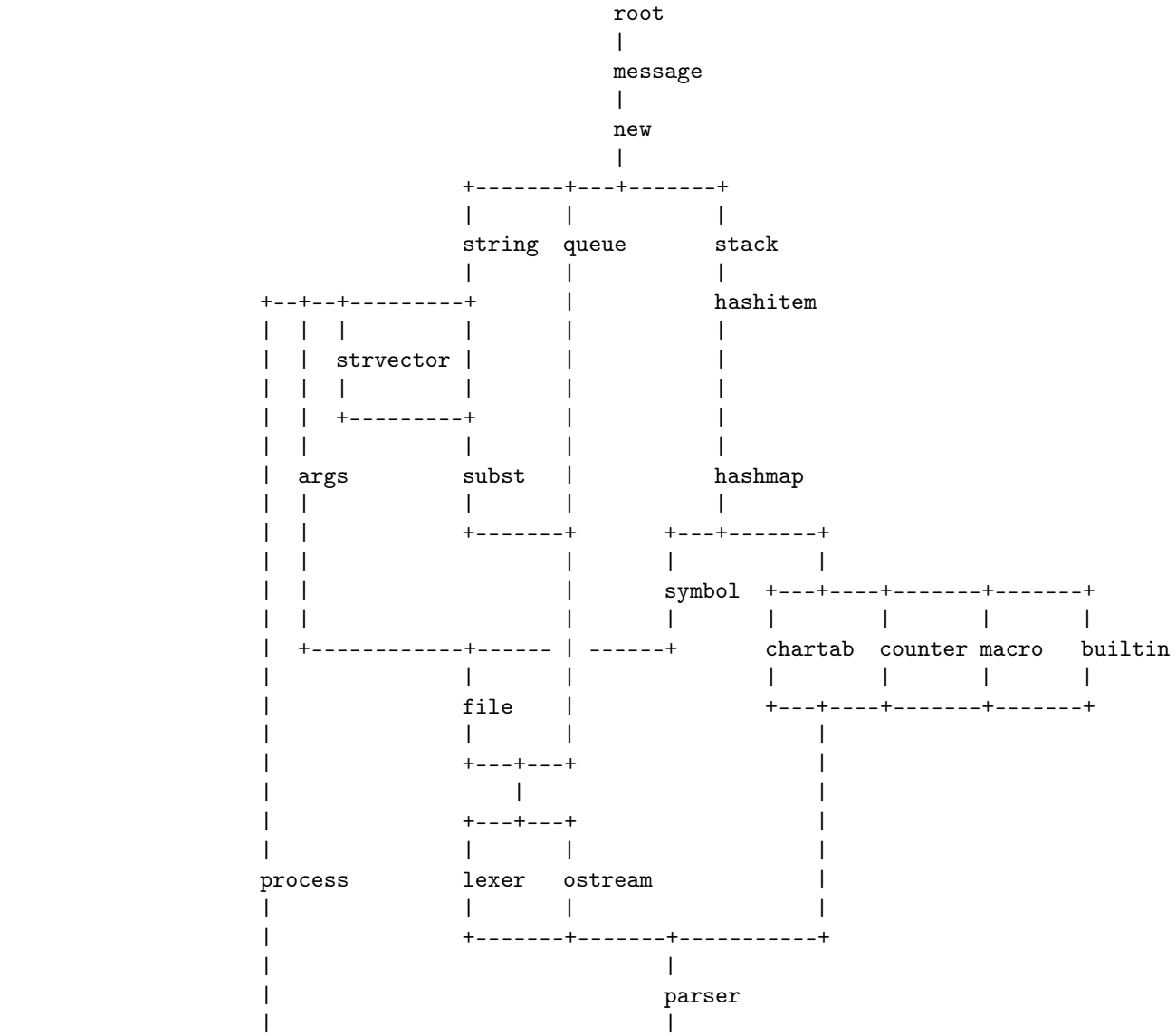
`Yodl`'s components show a strict hierarchical ordering. This allows the testing and development of components placed nearer to the component's tree without considering anything that's placed farther away.

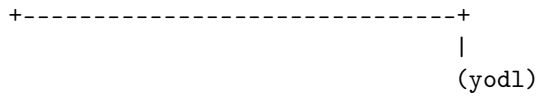
The following piece of 'ascii-art' shows the relationships for the `Yodl` program. The root of the tree starts at the top, at the `root` component. The tree can be read from the top to the bottom, where each horizontal line starts a level of components mentioned immediately below it, and each vertical route through the figure a series

of components whose functioning depend on at least the components mentioned earlier.

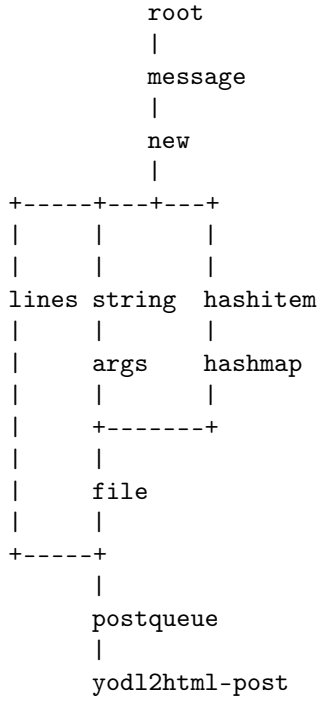
However, a more natural way to look at it is to start somewhere in the tree, and see what's envountered going up. Doing so, all components that are required are visited. Once the figure shows a

construction. This means that the horizontal line is not related to the vertical dependency crossing (but not touching) it.





A similar, albeit much simpler, tree can be drawn for `yodl-post`. Here is the organization of the components for the `yodl-post` program:



The source files of each component are organized as follows:

- All the files of a component are stored in a directory, named after the component. For example, the `counter` component is found in the directory

  ```
  yodl/counter
  ```

  containing all the (source) files that define that component.

- Each function is stored in a file of its own inside its component-directory. For example, the function `counter_value()` is defined in the source file `countervalue.c`.

- The file names are identical to the names of the functions, except for the fact that only lower case letters are used for the file names, and that the file names never use underscore characters.

- The `.h` header files declare the functions that can be used by other components. These functions are comparable to **C++**'s *public* members. Furthermore, these `.h` files define all structs and typedefs that are required for other

components to use a particular component. For example, the `component.h` header file may contain

```
#ifndef _INCLUDED_COUNTER_H_
#define _INCLUDED_COUNTER_H_

#include "../root/root.h"
#include "../hashmap/hashmap.h"

void        counter_add(HashItem *item, int add);   /* err  if no counter   */
bool        counter_has_value(int *valuePtr, HashItem *item);
Result      counter_insert(HashMap *symtab, char const *key, int value);
void        counter_set(HashItem *item, int value); /* err  if no counter   */
char const *counter_text(HashItem *item);       /* returns static buffer    */
int         counter_value(HashItem *item);      /* err  if no stack/item    */

#endif
```

- All functions declared in `.h` file start with the name of the component, and often contain an initial pointer to some `struct` containing the essential fields that are associated with that particular component. For example, most `counter_` functions have a `HashItem *` as their first argument, as a `HashItem` is normally used to store the details about a counter.
- The modifier `const` is used with pointers to indicate that the information pointed to by the pointer is 'owned' by the provider of that information. With parameters it indicates that the caller owns the information, and the function will not modify the provided info; with return types it indicates that the function 'owns' the returned information, which therefore may not be modified (or freed) by the caller of that function (e.g., `char const *counter_text`). The absence of `const` in combination with pointers indicates that the information pointed to by the pointer could, in principle, be modified by the code receiving the pointer value.
- Most components also show a `.ih` file, a so-called *internal header* file. The internal header declares 'internal support functions', not to be used by other parts of the software, and defines internal typedefs. Since they are an essential ingredient of the component, all these internal headers start to include the component's `.h` file, followed by the declarations of the 'private' functions. All these private functions start with abbreviated component names, like `co_` in the case of counters. Here is a possible implementation of the `counter.ih` internal header file:

```
#include "counter.h"

#include <stdio.h>

#include "../stack/stack.h"
#include "../message/message.h"
#include "../new/new.h"

Stack  *co_construct(int value);
```

```
Stack  *co_sp(HashItem *item, bool errOnFailure);
```

- The combination of `.h` and `.ih` files define the dependencies of the component in the component hierarchy. As can be seen, `counter` depends on `stack, message, new, hashmap` and `root`. The actual dependency listing may be a bit more complex, as some `.h` files themselves depend on other `.h` files. This is clearly visible in the `counter.h` file. The class hierarchy given earlier shows the final component dependencies.
- A `.h` file of a component `X` *never* includes a `.ih` file of component `Y`, but only the `.h` files of other components.

6.4 The token-producer 'lexer_lex()'

Tokens are produced by the lexical scanner. The function `lexer_lex()` produces the next token, which is always an element of the following set:

```
TOKEN_UNKNOWN,          /* should never be returned */

TOKEN_SYMBOL,
TOKEN_TEXT,
TOKEN_PLAINCHAR,        /* formerly: anychar */
TOKEN_OPENPAR,
TOKEN_CLOSEPAR,
TOKEN_PLUS,             /* it's semantics what we do with a +, not      */
                        /* something for the lexer to worry about       */

TOKEN_SPACE,            /* Blanks should be at the end                  */
TOKEN_NEWLINE,

TOKEN_EOR,              /* end of record: ends pushed strings           */
TOKEN_EOF,              /* at the end of nested evaluations/eof         */
```

In particular note the existence of a `TOKEN_EOR` token: this token indicates the end of a piece of text, a string, inserted into the input stream by the *parser*'s actions, when it calls `lexer_push_str()`. Such a situation occurs in particular when a macro is evaluated: having read a macro, and replacing its parameters `ARG1, ARG2, ... ARGn` by their respective argumentes, the resulting string is pushed back into the input stream by `lexer_push_str()`. This happens, e.g., inside the function `p_expand_macro()`. An excerpt from this function shows this call:

```
void p_expand_macro(register Parser *pp, register HashItem *item)
{
    ...
        if (argc)                           /* macro with arguments     */
            p_macro_args(pp, &expansion, argc);
        ...
```

```
            lexer_push_str(&pp->d_lexer, string_str(&expansion));
        ...
    }
```

The parser repeatedly calls the lexer's function `lexer_lex()`. This happens most dramatically inside the function `p_parse()`, defined by a mere single statement:

```
void p_parse(register Parser *pp)
{
    while ((*pp->d_handler[lexer_lex(&pp->d_lexer)])(pp))
        ;
}
```

Here, in a loop continuing until the handler indicates that the loop should terminate, `lexer_lex()` is called to produce the next token. The finite state automaton (FSA) implemented here is described in more detail in section 6.5.

Apart from here, `lexer_lex()` is called from four other locations inside the `parser` component:

- `parser_parlist()` repeatedly calls `lexer_lex()` to obtain all the tokens associated with a parameter list;
- `p_handle_default_newline()` repeatedly calls `lexer_lex()` to obtain all the tokens until all consecutive spaces and newlines are read. This is one of the handlers of the parser FSA 6.5;
- `p_no_user_macro()` calls `lexer_lex()` to determine whether a 'no user macro' has been detected;
- `p_plus_series()` calls `lexer_lex()` to determine whether a `+symbol` has been encountered.

So, `lexer_lex()` is the parser's 'window to the outside world'. The `lexer_lex()` function, however, is a fairly complex animal:

- `lexer_lex()`: returns next token. It calls `l_lex()` to retrieve the next character from the info waiting to be read;
- `l_lex()`: calls `l_nextchar()` to obtain the next token, and appends all char-tokens to the lexer's matched text buffer. Potential compound symbols (words, numbers) are combined by `l_compound()` and are then returned as `TOKEN_PLAINCHAR` or as a compound token like `TOKEN_IDENT`;
- `l_nextchar()`: calls `l_get()` to get the next character, and handles escape chars, including \ at eoln;
- `l_get()`: if there are no media left, `EOF` is returned. If there are media left, then `l_subst_get()` retrieves the next character, handling possible `SUBST` definitions. At the end of the current input buffer (memory buffer or file) `l_pop()` attempts to reactivate the previous buffer. If this succeeds, `EOR` is

returned, otherwise `EOF` is returned. So, the lexer is not able to switch between truly nested media, as in `EVAL()` calls, but is able to switch between nested buffers resulting from replacing macro calls by their definitions;

- `l_subst_get()`: calls `l_media_get()` to get the next char from the media. The next char is passed to subst_find() which is a FSA trying to match the longest `SUBST`. This may be done repeatedly, and eventually `subst_text()` either returns a substitution text, or the next plain character. A substitution text is pushed onto the lexer's media buffer. The next character returned is then the next one to appear at the lexer's media buffer;

- `l_media_get()`: If the current active source of information is a file, it returns the next character from that file or `EOF` if no such char is available anymore. If the current active source is a memory buffer then the next char from the buffer is returned. If the buffer is empty `EOF` is returned. The media buffer is a circular, self-expanding Queue.

6.5 The Parser's Finite State Automaton

The input files are parsed by the function `parser_process()`, which is called by `Yodl`'s `main()` function.

This processor pushes all files that were specified on the input in reverse order on the input stack, and then calls the support function `p_parse()` to process each of them in turn.

`p_parse()` is an very short function: it contains one `while` statement, repeatedly calling a *handler* appropriate with the next token returned by the lexical scanner. Therefore, the parser can be considered as a table driven finite state automaton (FSA).

The table itself is initialized in `parser/psetuphandlerset.c`, by the function `p_setup_handlerSet()`. It fills the two dimensional array `ps_handlerSet` with the address of the function that must be called for each combination of parser-state (as defined in the `HANDLER_SET_ELEMENTS` enum) in `parser/parser.h` and token that may be produced by the lexical scanner (as defined in the `LEXER_TOKEN` enum in `lexer/lexer.h`). Depending on the situation the parser encounters, it may point its pointer `d_handler` to a particular *row* in this table. Since the rows represent the parser's states, states can be switched easily by reassigning this pointer. This happens all the time. For example, when in `parsernameparlist.c` a name must be retrieved from a parameter list, it calls `parser_parlist(pp, COLLECT_SET)`, which function temporarily switches the parser's state to `COLLECT_SET`, returning the parameter list's contents. to its caller.

The functions whose addresses are stored in the various column-elements of the array `ps_handlerSet` are called *handler*. Most handlers are named `p_handle_<state>_<lextoken>()`, where `<state>` is the name of the associated parser state, and `<lextoken>` is the name of the appropriate lexical scanner token. For example, `p_handle_default_symbol()` is the handler that was designed for the situation where the parser is in its initial, or default, state, and the lexical scanner returns a `TOKEN_SYMBOL` token. Some handlers have more generic names, like `p_handle_unknown()`, which is some sort of emergengy exit, called when the parser doesn't know what to do with the received lexical scanner token (a situation which should, of course, not happen).

In version 2.00, the following handler functions are available:

- `p_handle_insert(Parser *pp)`: insert matched text
- `p_handle_default_eof(Parser *pp)`: return false
- `p_handle_default_newline(Parser *pp)`: series of \n's
- `p_handle_default_plus(Parser *pp)`: handle + series
- `p_handle_default_symbol(Parser *pp)`: handle all symbols
- `p_handle_ignore(Parser *pp)`: ignores token
- `p_handle_ignore_closepar(Parser *pp)`: handle openpar
- `p_handle_ignore_openpar(Parser *pp)`: handle openpar
- `p_handle_noexpand_plus(Parser *pp)`: handle + series
- `p_handle_noexpand_symbol(Parser *pp)`: handle executed symbols in NOEXPAND
- `p_handle_parlist_closepar(Parser *pp)`: handle closepar
- `p_handle_parlist_openpar(Parser *pp)`: handle openpar
- `p_handle_skipws_unget(Parser *pp)`: unget received text
- `p_handle_unexpected_eof(Parser *pp)`: EMERG exit
- `p_handle_unknown(Parser *pp)`: emergency exit

The parser has the following states:

COLLECT_SET
: retrieves parameter lists as they are encountered on the input. The parameter list is not processed in any way, and omits the surrounding parentheses. So, when entering this state (e.g., in the function `parser_parlist()`), a parameter list is completely consumed, but only its contents (and not its surrounding parentheses) become available. In fact, when entering a state, `p_parse()` can be called again to process the information in this state. Eventually a state will encounter some stopping signal (e.g., a non-nested close parenthesis in the collect-state results in `p_handle_parlist_closepar()` to return `false`, thus terminating `p_parse()`), terminating that particular state. The function `parser_parlist()` shows this process in further detail.

DEFAULT_SET
: In this state macros, builtins etc. are processed. For most of the tokens that can be returned by the lexical scanner `p_handle_insert()` is called.

  - When receiving EOF it tries to switch to the next file on the stack (or stop),
  - When receiving a symbol, it either handles them as plain symbols or as macros,
  - When receiving newlines they are handled (maybe merging them by calling a paragraph handler (if defined)),

- Series of + characters are handled
- All other tokens are inserted into the current output medium (which may be a file, but it may also be a memory buffer).

IGNORE_SET
In this state a parameter list is completely skipped. This state is used, for example, when processing `COMMENT()`.

NOEXPAND_SET
The contents of a parameter list is not expanded, but `CHAR` builtins *are* processed. In `Yodl` version 2.00 there is only one situation where this state (and its companion state NOTRANS_SET) is actively used: `Yodl`'s function `gram_NOEXPAND()` uses these states to retrieve the contents of a no-expanded or no-transed parameter list.

NOTRANS_SET
When the parser is in this state, a parameter list is inserted using the currently active insertion function (inserting to file or memory) It is identical to the NOEXPAND_SET state, but the character translation table is not used in the NOTRANS_STATE, whereas it is used in the NOEXPAND_STATE.

SKIPWS_SET
In this state all white-space characters are consumed. The lexical scanner merely returns the next non-whitespace character. This state is used, e.g., to skip the white space between multiple parameter lists when they are defined for macros.

6.6 Adding a new macro

With the advent of `Yodl` V 2.00, *raw macros files* are introduced. A raw macro file defines one macro, and *all* of its conversions. The raw macro files must be organized as follows:

```
<STARTDOC>
macro(name(arg1)(arg2)(etc))
(

    Description of the macro ‘name’, having arguments ‘arg1’, ‘arg2’,
    ‘etc’, each argument is given its own parameter list. The names of the
    arguments in this description should be chosen in such a way that they
    suggest their function or purpose. All macro descriptions starting
    with tt(<STARTDOC>) are included in both the ‘man yodlmacros’
    manpage and the description of the macro in the user guide. If this is
    not considered appropriate (e.g., tt(XX...()) macros are not described
    in these documents) then use tt(<COMMENT>) rather than
    tt(<STARTDOC>).
)
<>
DEFINEMACRO(name)(#)(
    statements of macro ‘name’ expecting ‘#’ arguments used by all
    conversions. This section is optional
<html>
```

```
    statements that should be executed by the HTML converter
<man ms>
    statements that should be executed by two converters. In this case,
    the ‘man’ and ‘ms’ converters
<else>
    statements that should be executed by all converters not explicitly
    mentioned above
<>
    statements of macro ‘name’ expecting ‘#’ arguments used by all
    conversions, having processed their specific statements.
    This section is also optional
)
```

When setting up these macro definitions, the <> tags must appear with the initial documentation section. It must also appear when at least one specific converter tag is used. For a macro which is converter independent, the macro definition doesn't contain these pointed-arrow tags.

When writing standard `Yodl` macros, each macro should be stored in a file `'name'.raw`, where `'name'` is the lower-case name of the macro. This file should then be kept in the `macros/rawmacros` directory. The `macros/build std` call then adds the macro (filtering only the required statements per conversion) to each of the standard conversion formats.

If the macro requires a counter or symbol, consider defining the counter or symbol in, respectively, `@counters` and `@symbols`. Furthermore, consider *pushing* and *popping* these 'variables', rather than plain assigning them, to allow other macros to use the variables as well. A case in point is the counter `XXone` which was added to the set of counters representing a *local counter*. Macros may *always* push `XXone` and pop `Xxone`, but should never reassign `XXone` before its value has been pushed. For `Yodl` version 2.00 only `XXone` was required, but other local counters might be considered useful in the future. In that case, `XXtwo`, `XXthree` etc. are used. For local symbold `XXs` prefixes are used: `XXsone`, `XXstwo`, etc.

6.7 The Yodl post-processor

With `Yodl` version 2.00 the old-style post-processor has ceased to exist. Also, the `.tt(Yodl)TAGSTART.` and `.tt(Yodl)TAGEND.` symbols no longer appear in `yodl`'s output.

Instead, a system using an *index* file was adopted. When converting information, `yodl` produces an output file and an associated *index* file. The index file defines *offsets* in the output file up to where certain actions are to be performed. Each line in the index file contains the required information of one *directive* for `yodlpost`. For example:

```
0 set extension man
53 ignorews
2112 verb on
2166 verb off
```

```
80007 ignorews
80065 copy
80065 mandone
```

Entries can be written into the index file using the `INTERNALINDEX` builtin function. This function has one argument: the information following the offset where it is called. So, there will be a `INTERNALINDEX(set extension man)` in the macro definitions for this particular conversion (obviously it is a `man` conversion. The particular `INTERNALINDEX` call is found in the standard `man.yo` macro definition file).

When `yodlmacros` is called, it processes the directives on the `idx` file in two steps:

- First, it reads all directives, and constructs a queue of actions to perform. During this phase it solves all references to, e.g., labels defined in the `s` processed by `yodl`. This queue is constructed by a `PostQueue` object, during its construction phase.

  Postprocessing is realized by a template-method design pattern-like construction in C.

  The algorithm proceeds as follows:

  Each element of the index file is read, and its keyword (the word following the offset) is determined. Then the 'construct' function associated with that keyword is called. The 'construct' functions return pointers to HashItem elements, which areprocessed by storing them either into the the symbol table or into the work-queue. The construct functions can use all `PostQueue, New, Message String Args` and `File` functions. Which function is actually called is determined in the file `yodlpost/data.c`, where the array `Task task[]` is initialized. `Task` structs have three elements:

  - `char const *d_key` points to the name of the keyword that triggers the corresponding `Task` struct;
  - `HashItem *(*d_constructor)(char const *key, char *rest)` points to the function that is called when the task struct is created.
  - `void (*d_handler)(long offset, HashItem *item)` points to the function that is called when the queue is processed.

- Then, when all commands are available, the queued commands are processed. For this, the appropriate 'handle' functions are called.

For example, when the `INTERNALINDEX(htmllabel ...)` is specified, the function `construct_label()` is called. This function receives a line line

```
432 label Overview
```

meaning that this label has been defined in offset 432 in the file generated by `yodl`. The `construct_label()` function now:

- Stores the current section number, the filecount and the sectionnumber in a HashItem.

- Stores the hashitem inside its hash-table.

Then, when the queue is processed, a reference to this label may be encountered. This is signalled by an `INTERNALINDEX(ref Overview)` call. In this case the `construct_ref()` function doesn't have to do much. Here it is the handler that's doing all the work:

- First it looks up the label in the symbol table. The label should be there, as a result of the earlier construction of the symbol table during the `postqueue_construct()` call.
- Then it copies the file written by `yodl` up to the offset mentioned in the the `ref` command.
- Then (since we're talking about an html-specific reference) the appropriate `<a href=...` command is inserted into the current output file.

When references are solved in text-files, the `INTERNALINDEX(txtref ...)` command is used. Here, `construct_ref()` can still be used, but a specific `handle_txt_ref()` function is required.

New postprocessing labels can be constructed easily:

- Add an element to the array `Task task[]` in `src/yodlpost/data.c`. For example, add a line like:

```
{"verb",            construct_verb,         handle_verb},
```

- Declare the functions in `yodlpost.h`:

```
HashItem *construct_verb(char const *key, char *rest);
void handle_verb(long offset, HashItem *item);
```

- The `construct_verb()` function receives the key (e.g., `verb`) and any information that may be available beyond the key as a trimmed line (not beginning or ending in white space). The construct function should return a pointer to a hashitem, which can be constructed by `hashitem_construct()`. This function should be called with the following arguments:
  - `VOIDPTR`;
  - a pointer to some text to be stored as the hashitem's key (use an empty string if nothing needs to be stored in a hashtable);
  - A pointer to the information associated with the key (use 0 if no information is used; use `(void *)intValue` to store an `int` value. Note that this is *not* `(void *)&intValue`: it is the value of the variable that is interpreted as a pointer here).
  - The function that handles the destruction of the value-information. Use `free` if some information was actually allocated and must be freed. E.g.,

```
hashitem_construct(VOIDPTR, "", new_str(rest), free);
```

Use `root_nop` if no allocation took place. E.g.,

```
hashitem_construct(VOIDPTR, "", (void *)s_lastLabelNr, root_nop);
```

Often the constructor doesn't have to do anything at all. In that case, initialize the `Task` element with the existing `construct_nop` function. E.g.,

```
{"drainws",         construct_nop,          handle_drain_ws},
```

- The `handle_verb()` function is called when the file produced by `yodl` is processed by `postqueue_process()`. This happens immediately after `postqueue_construct()`. The handler is called with two arguments:
  - Its first argument is the offset where the `INTERNALINDEX` call was generated. The handler should make sure that `yodl`'s output file is processed up to this offset. Not any further. If a simple copy is required the function `file_copy2offset()` is available. E.g.,

    ```
    file_copy2offset(global.d_out, postqueue_istream(), offset);
    ```

    Note its arguments: the output and input file pointers are available through, respectively, `global.d_out` and `postqueue_istream()`.
  - Its second argument is a pointer to the hashitem struct originally created by the matching `construct...()` function. The handler should *not* free the information it receives. The function `postqueue_process()` takes care of that.

  Examples of actual `construct...()` and `handle...()` functions can be found in `src/yodlpost`.