MlDoC

Version 1.0

OCaML Documentation System

Written by

Stefan Bosse (sbosse@physik.uni-bremen.de)

The *MlDoC* system is a powerfull but 'simple as needed' documentation tool for ML programming projects. With *MlDoC* , a document is divided in the document head, sections, subsections, units and subunits. Subunits can be used in all parts up, and the lowest structure elements are generic paragraph elements like lists or text attributes. **This document you are currently reading is of course written and printed with the *MlDoC* System.**
The table below shows an overview about *MlDoC* and it's capabilities.

DOCUMENT STRUCTURE					
Document head	**Sections**	**Subsections**	**Units**	**Subunits**	**Paragraph elements**
Title	Package	Module	ML-Function	Named Paragraph	Text Attributes
TOC	Program	C-Lib	ML-Type	Interface	Ordered List
			ML-Value	Example	Unordered List
			ML-Class		Table
			ML-Module		Name
			C-Interface		Link
	Generic S1	Generic S2	Generic S3	Generic S4	Special Interfaces
Intro	Intro	Intro	Intro	Intro	

It's not necessary to use all structure depths starting with the document head. For example you can start with the *ML-Function* unit in a small standalone document. The units are compareable with common manual pages.
Units are sub document headers, and subunits and introductions can be placed everywhere in the document.
The document structure is organized in the way *OCaML* structures libraries, modules and values, classes and sub modules. But *MlDoC* can also be used to provide documentation for programs, C libraries or other nice things you want to write about. For these purposes, generic sections, units and subunits can be used.
The *Special interface* element is used to show ML-function, value, type and sub module interfaces with an unique look, mostly embedded in the *Interface* subunit.
All document commands are preceeded with a dot and followed by one or two uppercase characters. This is a formatting language which is similar to *TROFF* macros. Each dot command must have an antidot command to close a partial environment. For example, an ordered list like

1. Line 1
2. Line 2
3. ...

is coded with the following commands:

```
.OL
  .LI Line 1 .IL
  .LI Line 2 .IL
  .LI ...  .IL
```

```
.LO
```

The antidot command is build with the reversed order of the two characters from the dot command. There is one excpetion: text attributes and some special commands. They all have only one uppercase character.

MlDoC dot commands

In this section, all available dot commands are described. They are sorted into these groups:

1. Section headers
2. Subsection headers
3. Unit headers
4. Subunit headers
5. Interfaces
6. Generic paragraph elements
7. Text elements and attributes

Sections together with subsections are used to structure the complete manual document.

AVAILABLE SECTIONS			
Name	**begin**	**end**	**Comment**
Package	`.PK`	`.KP`	Name argument required
Program	`.PR`	`.RP`	Name argument required
Generic Section S1	`.S1`	`.1S`	Name argument required

The *Name* command must follow the *section* command immediately. Here is a short example for a new *Package* section in the document:

PROGRAMMING EXAMPLE

```
.PK
  .NA Standard Library .AN
  ...
.KP
```

Packages can be ML-Libraries or other kinds of programming environments.

Subsections are used to group for example functions, values or other ML things. Commonly, a subsection is a part of a section.

Available subsections			
Name	**begin**	**end**	**Comment**
ML-Module	`.MD`	`.DM`	Name argument required
C-Library	`.CY`	`.YC`	Name argument required
Generic Subsection S2	`.S2`	`.2S`	Name argument required

Units are commonly known as manual pages. They are part of a subsection.

AVAILABLE UNITS			
Name	**begin**	**end**	**Comment**
ML-Function	`.FU`	`.UF`	Name argument required
ML-Type	`.TP`	`.PT`	Name argument required
ML-Value	`.MV`	`.VM`	Name argument required
ML-Class	`.CL`	`.LC`	Name argument required
ML-Module	`.MD`	`.DM`	Name argument required
C-Interface	`.CI`	`.IC`	Name argument required
Generic Unit S3	`.S3`	`.3S`	Name argument required

Subunits are special paragraphs. They can be used in all structure elements above.

AVAILABLE SUBUNITS (PARAGRAPHS)			
Name	**begin**	**end**	**Comment**
Named Paragraph	`.PA`	`.AP`	Name argument optional
Interface	`.IN`	`.NI`	Name argument optional
Example	`.EX`	`.XE`	Name argument optional
Preformatted text	.{	.}	
Generic Subunit S4	`.S4`	`.4S`	Name argument required

Except for a generic subunit, the name argument `.NA XXX .AN` is optional and can be used for a further description head.
The interface subunit is used to wrap function, value, type or other interfaces. This subunit is used in conjunction with the

- ML-Function
- ML-Value
- ML-Type (abstract types)
- ML-Structure
- ML-Module
- ML-Class
- C-Function/variables

special interfaces. See SPECIAL INTERFACES (p. 9) for more details. The *example* and the *preformatted* text subunits are used to display source code or other example lines in a new centered paragraph. The *example* subunit is put in a box. The *preformatted* subunit can spawn several pages.

The special interfaces are used to show ML or C functions, types, submodules and values. Only the content must be given by the user, not the alignment. Commonly, they are collected in an interface subunit.

AVAILABLE SPECIAL INTERFACES			
Name	**begin**	**end**	**Subarguments (∗=optional, #=multiple)**
ML-Function	`.IF`	`.FI`	Name, RetVal (#), Arg (#), Val (∗#)
ML-Value	`.IV`	`.VI`	Name, Arg (#)
ML-Type (type list)	`.IT`	`.TI`	Name, Arg (#)
ML-Structure (type)	`.IS`	`.SI`	Name, Arg (∗#)
ML-Module	`.IM`	`.MI`	Name, other specials
ML-Class	`.CS`	`.SC`	Name, Arg [∗#], Obj
ML-Class method	`.MT`	`.TM`	Name, Arg
C-Header	`.CH`	`.HC`	
C-Function	`.CF`	`.FC`	Name, Arg (#) , RetVal
C-Variable	`.CV`	`.VC`	Name, Arg

The followinf subarguments used only in special interfaces, except the *Name* command and the *Comment* command:

AVAILABLE SUBARGUMENTS			
Name	**Description**	**begin**	**end**
Name	Name of function,...	`.NA`	`.AN`
Comment	Placed before the special interface	`.(*`	`.*)`
RetVal	Return value of a function (uncurried subvalue)	`.RV`	`.VR`
Arg	Argument of a function (curried value)	`.AR`	`.RA`
Val	Value argument of a function (uncurried subvalue)	`.AV`	`.VA`
Obj	Class object	`.OB`	`.BO`

Either the curried value form **or** the uncurried value form must be used for the function interface. Here is an example for an interface with functions:

[*ret1* : int ∗ ≫*The first return argument*≪
ret2 : bool] = **myfun1** ≫*The second one.*≪
~arg1 : float → ≫*The first fun arg*≪
~arg2 : string list ≫*The second one.*≪

[*ret1* : bool] = **myfun2**
arg1 : float ∗
arg2 : string list

#include < **sys/io.h** >

int **cfun** (float arg1 ,
char arg2);

≫*The first value interface*≪

val **myval** : arg1 : string → ≫*The arg string*≪
arg2 : int →
retarg : float

The first and the second interfaces show a ML-Function, and the third one is a C-Function interface. This way to display ML-functions is different from the one commonly used in ML interfaces, but more readable. There is also a generic ML-value interface, shown in the fourth part of the example. In the second ML-function an uncurried value tuple is used for the function argument. Either the curried or the uncurried form must be used. The required programming code to get this beauty ML-interfaces is easy to create:

```
.IN
  .NA Function interfaces .AN
  .IF
    .NA myfun1 .AN
    .RV ret1:int .(* The first return argument .*) .VR
    .RV ret2:bool .(* The second one .*) .VR
    .AR ~arg1:float .(* The first fun arg .*) .RA
    .AR ~arg2:string list .(* The second one .*) .RA
  .FI
  .IF
    .NA myfun2 .AN
    .RV ret1:bool .VR
    .AV arg1:float .VA
    .AV arg2:string list .VA
  .FI
  .CH
    sys/io.h
  .HC
  .CF
```

```
    .NA cfun .AN
    .RV int .VR
    .AR float arg1 .RA
    .AR char arg2 .RA
  .FC
  .IV
    .(* The first value interface .*)
    .NA myval .AN
    .AR ~arg1:string .(* The arg string .*) .RA
    .AR ~arg2:int .RA
    .AR retarg:float .RA
  .VI
.NI
```

Always, labels must be marked with a leading tilde character. All other names appearing in value or function names are symbolic names and are displayed in a slatented font.
And an example for such an interface paragraph with ML-types and a ML-class:

PROGRAMMING INTERFACE

type **mytype** = Type_1
| Type_2
| Type_3

type **mytype** = { mutable a:int ;
b:float ;
c:int list }

exception **Failure of string**

class **myclass** :
a : int →
b : int
object
val mutable **speed** : *dir* : int list →
vl : int list →
?really : bool

val **align** : *up* : string list →
down : int list →
?really : bool

method private **up** : *v* : int →
name : string →
arg3 : int int list list →
()

end

```
module mymod
    sig
        type mytype =   Type_1
                      | Type_2
                      | Type_3
        val align : up : string list →
                    down : int list →
                    ?really : bool
        val mutable speed : dir : int list →
                            vl : int list →
                            ?really : bool
    end
```

which was produced with this source code:

```
.IN
  .IT
    .NA mytype .AN
    .AR Type_1 .RA
    .AR Type_2 .RA
    .AR Type_3 .RA
  .TI
  .IS
    .NA mytype .AN
    .AR .MU a:int .RA
    .AR b:float .RA
    .AR c:int list .RA
  .SI
  .CS
    .NA myclass .AN
    .AR a:int .RA
    .AR b:int .RA
    .OB
      .IV
        .NA .MU speed .AN
        .AR dir:int list .RA
        .AR vl:  int list .RA
        .AR ?really:  bool .RA
      .VI
      .IV
        .NA align .AN
        .AR up:string list .RA
        .AR down:  int list .RA
        .AR ?really:  bool .RA
      .VI
      .MT
        .NA .PV up .AN
        .AR v:int .RA
```

```
        .AR name:string .RA
        .AR ~arg3:int int list list .RA
        .AR () .RA
      .TM
    .BO
  .SC
  .IM
    .NA mymod .AN
    .IT
      .NA mytype .AN
      .AR Type_1 .RA
      .AR Type_2 .RA
      .AR Type_3 .RA
    .TI
    .IV
      .NA align .AN
      .AR up:string list .RA
      .AR down:  int list .RA
      .AR ?really:  bool .RA
    .VI
    .IV
      .NA .MU speed .AN
      .AR dir:int list .RA
      .AR vl:  int list .RA
      .AR ?really:  bool .RA
    .VI
  .MI
.NI
```

The class interface needs the class name, optional arguments, and the object block with values and methods. Of course, all possible special interfaces can be mixed, and the interface subunit is not required, but strongly recommended. Currently, only simple module signatures without functors are supported.

In this section, all generic paragraph elements like lists are described.
There are three different kinds of list you can use:

- Numbered lists (aka ordered lists)
- Unnumbered lists (aka unordered lists)
- Option lists (aka definition lists)

LISTS

Lists of different kind can be mixed and stacked to an arbitrary depth.

LISTS			
Name	**begin**	**end**	**Comment**
Ordered (numbered) List	`.OL`	`.LO`	
Unordered List	`.UL`	`.LU`	
Option List	`.PL`	`.LP`	First command in List item must be a Name.
List item	`.LI`	`.IL`	Used inside lists only.

1. 1
2. 2

Here is an example for an option list:

OPTIONS

-a
: This program option is used to show all available options. It's not needed in the server mode, but it's possible use this option in client mode.

-b
: This program option is used to build all available options.

-c
: This program option is used to convert all available options.

The required source code for this option list is shown below:

```
.PL
  .LI
    .NA -a .AN
    This program option is used to show all available options.
  .IL
  .LI
    .NA -b .AN
    This program option is used to build all available options.
  .IL
.LP
```

The *Option List* header can be changed with the *Name* argument, too.

Tables

There is support for simple tables, too. A *MlDoC* table is comparable with HTML tables. You need to specifiy the table body, an optional table head, table rows and in each table row the table column data.

Tables			
Name	**begin**	**end**	**Comment**
Table body	.TB	.BT	
NoRulers	.NR		Don't put boxes around the table cells.
Table head	.TH	.HT	Optional table head row.
Table row	.TR	.RT	
Table columns	.TC	.CT	Must be included between the row command.

Here is a small example for a generic table:

1	2
3	4

and the required source code:

```
.TB
  .TR
```

```
    .TC 1 .CT
    .TC 2 .CT
  .RT
  .TR
    .TC 3 .CT
    .TC 4 .CT
  .RT
.BT
```

The horizontal and vertical rulers are always present and can be removed with the *NoRulers* command. The `.NR` command must follow the *TableBody* command immediately. The above table looks then like:

1	2
3	4

Misc. elements:

Name	**begin**	**end**	**Description**
Name	`.NA`	`.AN`	Outside from special interfaces, this command emphasize names of functions, types or other important names.
Link	`.LK`	`.KL`	A link to another section or document

Name	**begin**	**end**	**Description**
Newline	`.NL`		Flush the current text paragraph and force a newline.
Indentmark	`.#`		Increase the current text indention. Only used in preformatted paragraphs like examples.
Preformatted text ('as is')	`. [`	`. ]`	Print inlined raw text. Ignore all dot commands.
Comments	`.(*`	`.*)`	Comments can be placed in *Arguments* .

See SPECIAL INTERFACES (p. 9) for comment examples and the place to use them. Use the *Newline* command only to finish the current text paragraph, and not to create space between paragraphs or other structure elements.
There are several text attribute commands controlling the appearence of text. All font styles must be switched off with the `.R` regular style command.

TEXT ATTRIBUTES	
Attribute	**command**
Boldface style	`.B`
Italic style	`.I`
Boldface and italic style	`.BI`
`Typewriter style`	`.T`
Romane style (default)	`.R`
Superscript $^{\text{textmode}}$	`.SP .PS`
Subscript $_{\text{textmode}}$	`.SB .BS`

TEXT STYLES AND SPECIAL TEXT

`.B`	`.R`	Bold text style
`.BI`	`.R`	Bold-Italic text style
`.I`	`.R`	Italic text style
`.T`	`.R`	Typewriter text style
`.S`	`.R`	Symbol text style [NA]
`.SB`	`.BS`	Subscript text mode
`.SP`	`.PS`	Superscript text mode
`. [`	`. ]`	'As-Is' text mode
`.NL`		Newline: End of paragraph

SPECIAL DOT COMMANDS

`.MU`		Special ML-attribute *mutable*
`.PV`		Special ML-attribute *private*
`.VT`		Special ML-attribute *virtual*
`.NR`		No rulers around tables and table cells
`.#`		White Space in *example* and *As-Is* environments
`.<<`	`.>>`	Include a *MlDoc* file at the current position
`.LK`	`.KL`	A document link

LIST COMMANDS

`.OL`	`.LO`	Numbered (ordered) list body
`.UL`	`.LU`	Unnumbered (unorderd) list body
`.PL`	`.LP`	Option (definition) list
`.LI`	`.IL`	A list item

TABLE COMMANDS

.TB	.BT	Table body
.TR	.RT	Table row
.TC	.CT	Table column
.TH	.HT	Optional table head

SPECIAL INTERFACES COMMANDS

.IF	.FI	ML-Function Interface
.IV	.VI	ML-Value Interface
.IT	.TI	ML-Type Interface
.IS	.SI	ML-Structure Interface
.IX	.XI	ML-Exception Interface
.IM	.MI	ML-Module Interface
.CH	.HC	C-Header Interface
.CF	.FC	C-Function Interface
.CV	.VC	C-Variable Interface
.CS	.SC	ML-Class Interface
.OB	.BO	ML-Object (Class) Interface
.MT	.TM	ML-Method (Class) Interface
.NA	.AN	Name argument
.AR	.RA	Curried value argument
.AV	.VA	Uncurried (tuple) value argument
.RV	.VR	Return argument of a function

SECTION COMMANDS

.PR	.RP	Program Section
.PK	.KP	Package Section

SUBSECTION COMMANDS

.MD	.DM	ML-Module
.CY	.YC	C-Library

UNITS OF A SUBSECTION (MANUAL PAGE) COMMANDS

`.FU`	`.UF`	ML-Function
`.MV`	`.VM`	ML-Value
`.TP`	`.PT`	ML-Type (generic)
`.CL`	`.LC`	ML-Class
`.CI`	`.IC`	C

SUBUNIT COMMANDS

`.IN`	`.NI`	Generic Interface Paragraph
`.EX`	`.XE`	Example Paragraph
`.{`	`.}`	Preformatted Paragraph
`.PA`	`.AP`	A named Paragraph
`.IO`	`.OI`	An Introduction Paragraph

GENERIC SECTION, SUBSECTION, UNIT, SUBUNIT COMMANDS

`.S1`	`.1S`	Generic Section
`.S2`	`.2S`	Generic Subsection
`.S3`	`.3S`	Generic Unit
`.S4`	`.4S`	Generic Subunit

First you must compile the package, consisting of the core module, the backend and the frontend modules. Here the steps needed for the case you use the generic OCaML system, version 3.XX:

```
ocamlc -c doc_core.ml
ocamlc -c doc_html.mli
ocamlc -c doc_html.ml
ocamlc -c doc_latex.mli
ocamlc -c doc_latex.ml
ocamlc -c doc_text.mli
ocamlc -c doc_text.ml
ocamlc -c doc_help.ml
ocamlc -a doc_core.cmo
    doc_html.cmo
    doc_latex.cmo
    doc_text.cmo
    doc_help.cmo -o mldoc.cma
```

You need the *Terminfo* interface file `terminfo.cmi` to be able to compile the *Doc_text* module. For *VamSyS* users:

```
vamcomp doc_core.ml
vamcomp doc_html.mli
vamcomp doc_html.ml
vamcomp doc_latex.mli
vamcomp doc_latex.ml
vamcomp doc_text.mli
vamcomp doc_text.ml
vamcomp doc_help.ml
vamar doc_core.cmo
    doc_html.cmo
    doc_latex.cmo
    doc_text.cmo
    doc_help.cmo -o mldoc.cma
```

PACKAGE: *mldoc*

The *mldoc* library is the heart of the *MlDoC* system. It's divided in the core module and various backend modules and some frontend modules.

DESCRIPTION

The core module of *MlDoC* . This module contains all basic functions and types used by all other frontend and backend modules.

Flow of processing string text:

1. Split the string text in atoms divided by blank spaces.
2. Translate these atoms in a structure tree.
3. Convert the structure tree into the desired output format.

First the interface of the main *structure_content* type and the the main *structure_block* :

PROGRAMMING INTERFACE

```
type structure_content =   S_Empty                 ≫Empty element ≪
                         | S_Text of (string list) ≫Generic text: basic element ≪
                         | S_Body                  ≫Document root ≪
                         | S_Title                 ≫Document title ≪
                         | S_TOC                   ≫Table of contents ≪
                         | S_Intro                 ≫An intro paragraph ≪
                         | S_Package               ≫Package=Library ≪
                         | S_Program               ≫A program description ≪
                         | S_S1                    ≫Generic named section ≪
                         | S_Module                ≫A ML-module ≪
                         | S_Clib                  ≫A C-Library ≪
                         | S_S2                    ≫Generic subsection ≪
                         | S_Function              ≫A function subsection ≪
                         | S_Type                  ≫A type subsection ≪
                         | S_Value                 ≫A value subsection ≪
                         | S_Class                 ≫A class subsection ≪
                         | S_C                     ≫C functions and variables ≪
                         | S_S3                    ≫Generic unit ≪
                         | S_Paragraph             ≫A named paragraph ≪
                         | S_Interface             ≫Generic Interface ≪
                         | S_Example               ≫Example preformatted paragraph ≪
                         | S_Preform               ≫Preformatted paragraph ≪
                         | S_S4                    ≫Generic subunit ≪
                         | S_Fun_Interface         ≫ML-Function interface ≪
                         | S_Val_Interface         ≫ML-Value interface ≪
                         | S_Type_Interface        ≫A ML-type definition ≪
                         | S_Exc_Interface         ≫Exception ≪
                         | S_Struc_Interface       ≫A type structure ≪
                         | S_C_Hdr_Interface       ≫C header interface ≪
                         | S_C_Fun_Interface       ≫C function interface ≪
                         | S_C_Var_Interface       ≫C variable interface ≪
```

```
| S_Class_Interface        (* A ML-Class *)
| S_Method_Interface       (* A ML method of a class *)
| S_Object_Interface       (* A ML object of a class *)
| S_Module_Interface       (* A ML module interface *)
| S_Attribute              (* Text or other attributes changes *)
| S_OList                  (* Ordered List *)
| S_UList                  (* Unordered List *)
| S_OpList                 (* Definition List *)
| S_List_Item              (* Unordered List item *)
| S_Table                  (* Table body *)
| S_TableHead              (* Table header *)
| S_TableRow               (* A new table row *)
| S_TableCol               (* Table column data *)
| S_TableNoRulers          (* Don't put boxes around the table cells *)
| S_Name                   (* Function, Type,... name *)
| S_Mutable                (* Mutable prefix *)
| S_Private                (* Provate prefix *)
| S_Virtual                (* Virtual prefix *)
| S_CurArg                 (* Argument of a curried list *)
| S_UnCurArg               (* Argument of an uncurried list:tuple *)
| S_RetArg                 (* Return argument of a function *)
| S_NL                     (* Newline *)
| S_TAB                    (* Text tabulation *)
| S_Link                   (* Link *)
| S_Comment                (* A comment *)

type structure_block = { mutable s_parent: structure_block option ;
                         mutable s_childs: structure_block list ;
                         mutable s_content: structure_content ;
                         mutable s_attr: structure_attr list ;
                         mutable s_line: int ;          (* For error tracking *)
                         mutable s_name: string ref }   (* For error tracking *)
```

A string can be converted to an atom list with the *atoms_of_text* function, and the *atoms_of_file* functions reads text from a file and convert it to an atom list:

PROGRAMMING INTERFACE

```
[ atom_list : string list ] = atoms_of_text
                                ~text : string

[ atom_list : string list ] = atoms_of_file
                                ~fname : string
```

The atom list can now be converted to a *struture_block* tree with the *tree_of_atoms* function. This function uses the *parse* generic text parser function. Finally, a *structure_block* tree can be printed in symbolic form with the *print_tree* function.

PROGRAMMING INTERFACE

```
[ unit ] = parse
            atom_list : string list →
            cur_block : structure_block

[ struc_tree : structure_block ] = tree_of_atoms
                                    ~atoms : string list

[ unit ] = print_tree
            structure_block
```

Sometimes it's usefull to convert only a part of the full document. For this purpose, all back end functions have the *section_names* list argument to tell the backend the current section environment. The *Doc_core* module defines the *section* structure and the *section_names* types. The structure is only for internal use in the backend.

PROGRAMMING INTERFACE

```
type section_names =   Sec_s1 of string
                     | Sec_s2 of string
                     | Sec_s3 of string
                     | Sec_s4 of string
                     | Sec_package of string
                     | Sec_program of string
                     | Sec_module of string
                     | Sec_function of string
                     | Sec_type of string
                     | Sec_val of string
                     | Sec_class of string
                     | Sec_cint of string

type section = { mutable sec_parent: section ;
                 mutable sec_childs: section list ;
                 mutable sec_name: string ;
                 mutable sec_type: string }
```

A short example follows to show the processing flow to translate a help text file to a *structure_block* tree.

PROGRAMMING EXAMPLE

```
let al = atoms_of_file "mldoc.man" ;;
let st = tree_of_atoms al ;;
html_of_tree ds [HTML_single_doc "mldoc_man.html"] [];;
tex_of_tree ds [TEX_doc_name "mldoc.tex";
    TEX_color;
    TEX_link_ref;
  ] [];
text_of_tree ds [TEXT_doc_name "mldoc.txt"] [];
```

DESCRIPTION

This is the HTML backend module used to convert *MlDoC* documents in HTML-4 (transitional) format. This *html_of_tree* function is used to convert the *structure_block* graph to HTML output.

PROGRAMMING INTERFACE

[unit] = **html_of_tree**
~ds : structure_block →
~options : html_options list →
~sections : section_names list

The *html_of_tree* function needs three arguments:

ARGUMENTS

ds
This is the structure block tree previously generated with the *atoms_of_* function family.

options
Several options can control the converting behaviour and the output. Available options:

`HTML_single_doc of string`	Only create a single HTML file instead of a collection of files, each for one section. The string is the file name.

sections
To convert only a part of a manual, for example a module like this, some informations are needed about sections above this section.

MODULE DEPENDENCIES:

- *Doc_core*

DESCRIPTION

This is the Tex backend module used to convert *MlDoC* documents in Plain-TeX/Latex format. This *tex_of_tree* function is used to convert the *structure_block* graph to TeX output.

PROGRAMMING INTERFACE

[unit] = **tex_of_tree**
~ds : structure_block →
~options : tex_options list →
~sections : section_names list

The *tex_of_tree* function needs three arguments:

ARGUMENTS

ds
This is the structure block tree previously generated with the *atoms_of_* function family.

options
Several options can control the converting behaviour and the output. Available options:

`TEX_doc_name of string`	The name of the output file. The default setting is `manual.tex` .
`TEX_head_inline`	Don't perform a page break on section boundaries.
`TEX_color`	Ouput colored manual pages. Special color commands for *dvips* are created. Default is color off.
`TEX_no_toc`	Don't print a table of content. Default is print a TOC.
`TEX_link_ref`	Generate links in the manual (page references). Default is no links.

sections
To convert only a part of a manual, for example a module like this, some informations are needed about sections above this section.

LATEX PACKAGE DEPENDENCIES

The following packages are used by the *Doc_latex* module:

- `newcent`
- `pifont`
- `color[dvips]`

Most common LaTeX/TeX distributions are shipped with these packages.

MODULE DEPENDENCIES:

- *Doc_core*

DESCRIPTION

This is the ASCII-Text backend module used to convert *MlDoC* documents in generic Text format. This *text_of_tree* function is used to convert the *structure_block* graph to text output.

PROGRAMMING INTERFACE

```
[ unit ] = text_of_tree
              ~ds : structure_block →
              ~options : text_options list →
              ~sections : section_names list
```

The *text_of_tree* function needs three arguments:

ARGUMENTS

ds
This is the structure block tree previously generated with the *atoms_of_* function family.

options
Several options can control the converting behaviour and the output. Available options:

`TEXT_doc_name of string`	The output file name if any. Without this option, the output is printied to the current standard out channel.
`TEXT_terminal`	Enhanced text ouput with special terminal control command (underlined...). Default is no special commands.
`TEXT_notoc`	Don't print a table of content.

sections
To convert only a part of a manual, for example a module like this, some informations are needed about sections above this section.

MODULE DEPENDENCIES:

- `Terminfo` (OCaML)
- `Doc_core`

DESCRIPTION

This is the *Help* frontend. It can be used to handle *MlDoc* documents with various backend modules. First, a document must be loaded. Either, you use the *atoms_of_file* and the *tree_of_atoms* functions from *Doc_core* module, finally loaded with *help_load* function, or you use the *help_file* function to load a document.

PROGRAMMING INTERFACE

```
type h_sec = { mutable h_sec_keywords: string list ;
               mutable h_sec_ds: structure_block ref ;
               mutable h_sec_env: section_names list ;
               mutable h_type: string }

type s_help = { mutable h_sections: h_sec list ;
                mutable h_subsections: h_sec list ;
                mutable h_units: h_sec list ;
                mutable h_main: structure_block }

[ help : s_help ] = help_of_doc
        ~ds : structure_block

[ unit ] = help_load
        ~ds : structure_block

[ unit ] = help_file
        ~fname : string

type help_Device =   Help_TTY      ≫Print to stdout≪
                   | Help_ASCII    ≫Print to file: ASCII text≪
                   | Help_HTML     ≫Print to file: HTML≪
                   | Help_LATEX    ≫Print to file: Latex≪

[ unit ] = help_dev
        ~dev : help_device

[ unit ] = help
        ~name : string
```

The *help_dev* function can be used to change the output device. Default is the *Help_TTY* device. The *help* function uses the keyword or name string to search the help database. The search results will be shown.

PROGRAMMING EXAMPLE

```
let at = atoms_of_file "mldoc.man" in
let ds = tree_of_atoms at in
help_load ds;
```

MODULE DEPENDENCIES

- *Doc_core*
- *Doc_html*
- *Doc_latex*
- *Doc_text*

TABLE OF CONTENT