



## Working with EIDB

Erich Frühstück

**E**rich **F**rühstück **E**ntwicklungs **U**mgebung

Copyright (C) 2006 Erich Frühstück.  
This documentation is part of EFEU.

EFEU is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

EFEU is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with EFEU; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

# Preface

This manual describes working with EDB data files. The work on this documentation is still in progress.

Erich Frühstück  
Wördern, March 2006

# Contents

- 1 Basics** **3**
- 1.1 The EDB data format . . . . . 3
- 1.2 The EDB interface . . . . . 3
  
- Commands** **4**
- edb – copy efu data files . . . . . 5
- edbcats – concatenation of efu data files . . . . . 7
- edbjoin – joining efu data files . . . . . 9
- esh – EFEU command interpreter . . . . . 11
- md2edb – convert data cube to efu data files . . . . . 24

# Chapter 1

## Basics

1.1 The EDB data format

1.2 The EDB interface

# Commands

**NAME**

edb – copy efeu data files

**SYNOPSIS**

```
edb [ --help[=type] ] [ --version ] [ --info[=entry] ] [ --debug[=mode] ] [ --verbose ] [ -1 ]
    [ --fast ] [ -9 ] [ --best ] [ -I dir ] [ -E expr ] [ -F file ] [ -T title ] [ -p mode ] [ -d ]
    [ -n lim ] [ -f def ] [ -s file ] [ --script file ] in [ out ]
```

**DESCRIPTION**

The following options and arguments are accepted from command **edb**:

**--help**[=*type*]

create command usage. The optional parameter *type* determines the formatting and output of the description.

term

display on terminal (default)

raw

output raw data for **efeudoc**

man

output nroff/troff source for **man**

lp send postscript data to **lpr**

**--version**

show version information

**--info**[=*entry*]

show command information

**--debug**[=*mode*]

set debug level for command See for details.

**--verbose**

set debug level to **.info**.

**-1, --fast**

use fast compression by writing with **gzip**.

**-9, --best**

use best compression by writing with **gzip**.

**-I *dir***

append *dir* to the search path for script headers. The default setting is `"/home/efeu/www/efeu-3.3-1/`

**-E *expr***

evaluate expression *expr*

**-F *file***

execute command lines in file *file*

**-T *title***

set title of output file

**-p *mode***

output mode

**-d**

print data without header

- n** *lim*  
Limit of output data lines
- f** *def*  
apply filter *def* on input file
- s** *file*, **--script** *file*  
apply filter script *file* on input file
- in* input file name
- out* output file name

## ENVIRONMENT

- APPLPATH**  
path for configuration files.
- LANG**  
locale information

## COPYRIGHT

Copyright (C) 2004 Erich Frühstück

**NAME**

edbcats – concatenation of efeu data files

**SYNOPSIS**

```
edbcats [ --help[=type] ] [ --version ] [ --info[=entry] ] [ --debug[=mode] ] [ --verbose ] [ -1 ]
[ --fast ] [ -9 ] [ --best ] [ -I dir ] [ -E expr ] [ -F file ] [ -T title ] [ -p mode ] [ -n lim ]
[ -a list ] [ --pre_filter list ] [ -b list ] [ --post_filter list ] [ -m cmp ] [ --merge cmp ]
[ -s cmp ] [ --sort cmp ] [ -o out ] list
```

**DESCRIPTION**

The following options and arguments are accepted from command `edbcats`:

`--help[=type]`

create command usage. The optional parameter *type* determines the formatting and output of the description.

term

display on terminal (default)

raw

output raw data for `efeudoc`

man

output nroff/troff source for `man`

lp

send postscript data to `lpr`

`--version`

show version information

`--info[=entry]`

show command information

`--debug[=mode]`

set debug level for command See for details.

`--verbose`

set debug level to `.info`.

`-1, --fast`

use fast compression by writing with `gzip`.

`-9, --best`

use best compression by writing with `gzip`.

`-I dir`

append *dir* to the search path for script headers. The default setting is `"/home/efeu/www/efeu-3.3-1/`

`-E expr`

evaluate expression *expr*

`-F file`

execute command lines in file *file*

`-T title`

set title of output file

`-p mode`

output mode

- n** *lim*  
Limit of output data lines
- a** *list*, **--pre\_filter** *list*  
apply filter *def* on every input file
- b** *list*, **--post\_filter** *list*  
apply filter *def* on concatenated files
- m** *cmp*, **--merge** *cmp*  
mischts die Dateien mit der Vergleichsfunktion *cmp*.
- s** *cmp*, **--sort** *cmp*  
sortiert die Dateien mit der Vergleichsfunktion *cmp* vor dem mischen/verknüpfen.
- o** *out*  
output file name
- list* list of input files

## ENVIRONMENT

- APPLPATH**  
path for configuration files.
- LANG**  
locale information

## COPYRIGHT

Copyright (C) 2004 Erich Frühstück

**NAME**

edbjoin – joining efeu data files

**SYNOPSIS**

```
edbjoin [ --help[=type] ] [ --version ] [ --info[=entry] ] [ --debug[=mode] ] [ --verbose ] [ -1 ]
        [ --fast ] [ -9 ] [ --best ] [ -E expr ] [ -F file ] [ -T title ] [ -p mode ] [ -n lim ] [ -j arg ]
        [ -a def ] [ -b def ] [ -f def ] db1 db2 [ out ]
```

**DESCRIPTION**

The following options and arguments are accepted from command `edbjoin`:

- `--help[=type]`  
create command usage. The optional parameter *type* determines the formatting and output of the description.
- term  
display on terminal (default)
- raw  
output raw data for `efeudoc`
- man  
output nroff/troff source for `man`
- lp send postscript data to `lpr`
- `--version`  
show version information
- `--info[=entry]`  
show command information
- `--debug[=mode]`  
set debug level for command See for details.
- `--verbose`  
set debug level to `.info`.
- `-1, --fast`  
use fast compression by writing with `gzip`.
- `-9, --best`  
use best compression by writing with `gzip`.
- `-E expr`  
evaluate expression *expr*
- `-F file`  
execute command lines in file *file*
- `-T title`  
set title of output file
- `-p mode`  
output mode
- `-n lim`  
Limit of output data lines
- `-j arg`  
sets the join parameter to *arg*

- a** *def*  
    applay filter *def* on first input file
- b** *def*  
    applay filter *def* on second input files
- f** *def*  
    applay filter *def* on output file
- db1***  
    first data base
- db2***  
    second data base
- out*** output file name

## ENVIRONMENT

- APPLPATH**  
    path for configuration files.
- LANG**  
    locale information

## COPYRIGHT

Copyright (C) 2005 Erich Frühstück

**NAME**

esh – EFEU command interpreter

**SYNOPSIS**

```
esh [ --help[=type] ] [ --version ] [ --info[=entry] ] [ --debug[=mode] ] [ --verbose ] [ -I dir ]
    [ -D name=val ] [ -c string ] [ -E ] [ file ] [ arg(s) ]
```

**DESCRIPTION**

The command `esh` evaluates scripts in the syntax of the EFEU interpreter language. The syntax of the language is similar to C/C++. If you are familiar with this language(s), you would easily learn to use this interpreter.

`esh` accepts comments in the style of C/C++ and uses a preprocessor similar to C/C++. See later in this document. Lines starting with `#!` are not interpreted by `esh`. If the script is executable and the first line is

```
#!path
```

where *path* is the full path name of the command `esh`, you can use it like an ordinary command. I prefer the following variation, which is independent from the installation place of `esh`:

```
#!/usr/bin/env esh
```

Expressions are terminated either by a semicolon or a linefeed, tabs and spaces are skipped. An expression may also end if there is no right operator following a term. In this case and if the next character is a punctuation character, it is used as termination key, else a space is used. On some places, e.g. inside the argument list of a function, a linefeed does not terminate the expression and is skipped like tabs or spaces.

In the outermost level (outside any block or function body) every statement is evaluated immediately after parsing. If an expression is not terminated by a semicolon, the result is written to standard output followed by the terminating character.

For example: The line

```
3 * 5 4 + 7 $ 2 - 1; 4 + 1
```

is split into the 4 expressions

```
3 * 5 terminated by space
4 + 7 terminated by $
2 - 1 terminated by ;
4 + 1 terminated by linefeed
```

and the output is

```
15 11$5
```

in outermost level.

If `esh` is called without script name or if the script name is a single minus, commands are read from standard input. If standard input and standard output is connected to a terminal, `esh` runs interactive and `readline` is used for reading lines from terminal. Readline control keys are active and `!` at beginning of a new line is used as control key to run history and system commands.

The use of `readline` in interactive mode and the automatic display of results in the outermost level allows to use `esh` as a comfortable desk calculator.

The complete EFEU interpreter language is implemented with C library functions. `esh` is a simple command which uses this functions. The interpreter shares data pointers directly with C functions. You can add your own functions and types to the interpreter. So it is easy to use this language for configuration files or to test functions with it.

If EFEU is build with shared libraries (e.g. on Linux) you can expand `esh` at run time. If shared libraries are not available, you can take a copy of `esh.c` and add your extensions there.

## Options

Options placed after the script name are interpreted by the script. The options `-?` and `--help` show you the syntax of the script.

The following options and arguments are accepted by `esh`:

- `--help[=type]`  
create command usage. The optional parameter *type* determines the formatting and output of the description.
  - `term`  
display on terminal (default)
  - `raw`  
output raw data for `efeudoc`
  - `man`  
output nroff/troff source for `man`
  - `lp` send postscript data to `lpr`
- `--version`  
show version information
- `--info[=entry]`  
show command information
- `--debug[=mode]`  
set debug level for command See for details.
- `--verbose`  
set debug level to `.info`.
- `-I dir`  
append *dir* to the search path for script headers. The default setting is `"/home/efeu/www/efeu-3.3-1/`
- `-D name=val`  
define macro *name* with value *val*
- `-c string`  
process commands from *string*.
- file* name of the script file.
- arg(s)*  
script parameters.
- `-E` preprocess only, do not evaluate commands.

## PREPROCESSOR

The EFEU interpreter language uses a preprocessor similar to C. The preprocessor is built in as a filter and can be used independent of the interpreter. The preprocessor evaluates the input per

line and not per file. So you can create or change variables which may be used later in conditional directives.

### Including files

Files are included by the `#include` directive. In the first step, macro substitution is performed on the rest of the line starting with `#include`. Afterwards, the following cases are possible:

`#include <name>`

This variant searches files in a list of directories, specified by the variable `IncPath`. Characters after `<` are silently ignored.

`#include expr`

First, `expr` is evaluated and converted to string. If the resulting string does not start with `<`, the current directory is searched before any other directory in `IncPath`.

`#include "name"`

This is only a special form of the above clause. The file `name` is first searched in the current directory.

In esh, the following construction is legal:

```
str header = paste("/", "SubDir", "MyHeader");
#include "<" + header + ">"
```

The variable `header` is defined in the outermost level, so it is immediately executed and can be used in the following `#include` directive. Adding `<` and `>` avoids searching the current directory (if `IncPath` does not include the current directory).

### Conditionals

A conditional begins with a 'conditional directive': `#if`, `#ifdef` or `#ifndef` and ends with `#endif`. A conditional block may contain `#elif` and `#else` directives. Conditional blocks may be nested.

The directives

`#ifdef name`

and

`#ifndef name`

are used to test macro definitions. No macro substitution is performed on this directive lines.

The simplest form of a conditional is:

```
#if expr
Input lines seen if expr is true.
#endif
```

A more complex conditional may look like this:

```
#if expr1
Input lines seen if expr1 is true.
#elif expr2
Input lines seen if expr2 is true and expr1 is false.
#else
Input lines seen if neither expr1 or expr2 is true.
#endif
```

As seen in the section ‘Including files’, you can use any variable or function in *expr* previously declared in the outermost level.

## Macros

Macros are defined with the `#define` directive. It has two forms:

`#define name replacement`  
defines a macro without arguments.

`#define name(args) replacement`  
defines a macro with arguments. The left parenthesis must follow immediately the name of the macro.

The name of a macro must start with an alphabetic or underline character and may contain only alphanumeric or underline characters.

In `esh` macros are rarely used. In most of all places, variables and functions are the better solution. Normally they are only used to protect header files for multiple inclusions.

A macro could be removed with the `#undef` directive.

## EXPRESSIONS

Constants and variables could be joined with operators to single expressions.

The next tables show the available operators of `esh`. They are sorted by descending priority. Operators not separated by a line have the same priority.

Prefix operators		
<code>::</code>	global	<code>::name</code>
<code>++</code>	pre increment	<code>++lvalue</code>
<code>--</code>	pre decrement	<code>--lvalue</code>
<code>~</code>	complement	<code>~expr</code>
<code>!</code>	not	<code>!expr</code>
<code>-</code>	unary minus	<code>-expr</code>
<code>+</code>	unary plus	<code>+expr</code>
<code>{</code>	list grouping	<code>{ expr [, expr ] }</code>
<code>(</code>	grouping	<code>( expr )</code>
<code>[</code>	expression	<code>[ expr ]</code>
<code>()</code>	cast (type conversion)	<code>(type) expr</code>

The list grouping operator creates a list of values. In opposite to C/C++, the use of this operator is not restricted to assigning data in variable declarations.

The expression operator parses an expression without evaluation. This expression may be stored in a variable or passed as function argument for later evaluation.

Postfix operators		
++	post increment	<i>lvalue++</i>
--	post decrement	<i>lvalue--</i>
::	scope resolution	<i>type::name</i>
::	variable selection	<i>vartab::name</i>
.	member selection	<i>expr.name</i>
[]	sub scripting	<i>expr[expr]</i>
()	function call	<i>expr(list)</i>

In esh, you have access to a table of variables and you can create your own variable tables. The scope resolution operator is used to get a member of this table. You can apply the operator to any type, that could be converted in the type `VarTab`.

Arithmetic operators		
*	multiply	<i>expr * expr</i>
/	division	<i>expr / expr</i>
%	modulo (remainder)	<i>expr % expr</i>
+	add (plus)	<i>expr + expr</i>
-	subtract	<i>expr - expr</i>
<<	shift left	<i>expr &lt;&lt; expr</i>
>>	shift right	<i>expr &gt;&gt; expr</i>

Comparison operators		
<	less than	<i>expr &lt; expr</i>
<=	less than or equal	<i>expr &lt;= expr</i>
>	greater than	<i>expr &gt; expr</i>
>=	greater than or equal	<i>expr &gt;= expr</i>
==	equal	<i>expr == expr</i>
!=	not equal	<i>expr != expr</i>

Bit wise operators		
&	bit wise AND	<i>expr &amp; expr</i>
^	bit wise exclusive OR	<i>expr ^ expr</i>
	bit wise inclusive OR	<i>expr   expr</i>

Logical operators		
&&	logical AND	<i>expr &amp;&amp; expr</i>
	logical OR	<i>expr    expr</i>

The right operand of a logical operator is only evaluated, if the resulting value is not determined by the left operand. So in

```

false && expr
true || expr

```

the right operand *expr* is never evaluated.

Conditional and range operator		
<code>? :</code>	conditional operator	<code>cond ? expr1 : expr2</code>
<code>:</code>	range operator	<code>start : end [ : step ]</code>

The range operator creates a list of variables, starting by *start* and ending by *end*. For numbers, the value of *step* must be positive. The default value is 1. Wrong use of this operator may result in an infinite loop. There is no range operator in C/C++.

Assign operators		
<code>=</code>	simple assignment	<code>lvalue = expr</code>
<code>*=</code>	multiply and assign	<code>lvalue *= expr</code>
<code>/=</code>	divide and assign	<code>lvalue /= expr</code>
<code>%=</code>	modulo and assign	<code>lvalue %= expr</code>
<code>+=</code>	add and assign	<code>lvalue += expr</code>
<code>-=</code>	subtract and assign	<code>lvalue -= expr</code>
<code>&lt;&lt;=</code>	shift left and assign	<code>lvalue &lt;&lt;= expr</code>
<code>&gt;&gt;=</code>	shift right and assign	<code>lvalue &gt;&gt;= expr</code>
<code>&amp;=</code>	AND and assign	<code>lvalue &amp;= expr</code>
<code>^=</code>	exclusive OR and assign	<code>lvalue ^= expr</code>
<code> =</code>	inclusive OR and assign	<code>lvalue  = expr</code>

Assign operators are right associative.

List separator		
<code>,</code>	list separator	<code>expr , expr</code>

The comma `,` in esh is used as list separator like python, not as comma operator like C/C++. So `a, b, ..., n` returns a list containing `a, b, ..., n`.

If you do not use the return value (in the most common use), there is no difference between the comma operator and the list separator.

## Loops

`while (cond) expr`

As long as `cond` is true, `expr` is executed.

`do expr while (cond)`

The expression `expr` is executed and repeated as long as `cond` is true.

`for (a; cond; b) expr`

First, `a` is evaluated. As long as `cond` is true, `expr` and then `b` is evaluated. Either `a`, `cond` or `b` may be omitted, `a` and `b` must be simple expressions.

`for (name in list) expr`

The representative `name` is set to each element of the list in turn, and `expr` is evaluated each time. If `list` consists of a single element and its type is convertible to `List_t`, the result of the conversion is used instead.

In any case of loop, the statement `break` breaks out of the loop and the statement `continue` starts the next cycle.

## Conditionals

```
if (cond) expr1
```

If the expression *cond* is true, *expr1* is executed.

```
if (cond) expr1 else expr2
```

If the expression *cond* is true, *expr1* is executed, else *expr2* is executed.

### Switch statement

The syntax of a `switch` statement is:

```
switch (expr)
{
  label:
      cmdlist
  label:
      cmdlist
  ...
}
```

where *label* may be `case val` or `default`. The expression *val* is evaluated on parsing and not on executing the switch statement. The value of *expr* is compared with all labels in order of definition. If the comparison is true, all following statements until `break`, `continue`, `return` or the end of the switch block is reached, are executed. If none of the labels compares with *expr*, all statements after `default` (if present) are executed.

In opposite to C, any data type is allowed in the switch statement as long as the operator `==` is defined. In particular you can use strings in switch statements.

### Grouping

Braces are used to group expressions to a block. A block has no return value. Every block has two tables of variables. The less visible table is created by parsing the command lines, the more visible table on evaluating the block. Every expression following the keyword `static` is executed immediately after parsing. So type declaration with the prefix `static` creates variables in the less visible table. The use of `static` is not restricted to declarations.

## DECLARATIONS AND CONSTANTS

In opposite to other interpreter languages, variables must be declared like C/C++ before use. A declaration may be placed anywhere in the source and has a return value (the value of the declared variable).

For example:

```
int x;
double a, b;
x = (int y = 5);
```

declares first the integer variable *x* and the double variables *a* and *b*. Afterwards the integer variable *y* is declared with value 5 and the result (the value 5) is assigned to *x*. It is allowed to declare a variable more than once with the same type. All but the first declarations are converted to a assignment statement.

Every predefined data type in the interpreter language has a corresponding data type in C. The EFEU interpreter language does not have pointers, but data types may be represented by pointer types in C.

The interpreter distinguishes between lvalues and constants. An lvalue is anything, that could stand on the left side of an assignment. Typical lvalues are variables. The result of an expression or a function call may be an lvalue or not.

### Integral types

Like in C/C++, there are several different integral types. The interpreter supports the exact-width integer types as defined in C99. They are `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t`. The following table shows additional integral types and their representation in C.

esh type	C type
<code>bool</code>	<code>int</code>
<code>int</code>	<code>int</code>
<code>unsigned</code>	<code>unsigned int</code>
<code>varint</code>	<code>int64_t</code>
<code>varsize</code>	<code>uint64_t</code>

The keyword `unsigned` is a type name and not a type qualifier like in C. The data types `varint` and `varsize` have a variable length binary representation in data files.

The syntax of integral constants is like C/C++. The keywords `true` (integral value 0) and `false` (integral value 1) are used for boolean values.

### Floating number types

Esh uses the types `float` and `double` like C/C++. Floating number constants are always of type `double`. All arithmetic is done by `double`, `float` is defined for completeness and to save space for large data fields, where precision is not so important.

### Characters and strings

A character in esh has the type `char` and its code value is considered as unsigned.

Strings are completely different implemented in `esh` than in C. They are not fields of type `char`, they have the data type `str`. If you assign a string to a value or use it as function argument, the whole string (not the address) is copied. Copying strings is always done with memory allocation and there is a built in garbage collection for it (and generally for all dynamic allocated objects).

Character constants are delimited by single quotes, string constants by double quotes. The backslash is used as escape character like in C.

String constants may contain linefeed. In esh, two strings next to each other are not concatenated like in C/C++. You need the add operator `+` to do this.

For long string constants there exists the keyword `string`, which is used in the following form:

```
string !
  contents of string
!
```

There must be a newline after `!` in the starting line and `!` must be the first character in the last line. A so defined string always contains a linefeed at end. The backslash is no longer used as escape character with one exception: protecting a `!` at beginning of line to be interpreted as string termination. This construction of strings may be used anywhere inside an expression.

Comments are skipped and preprocessor directives are interpreted inside this string definitions.

For example: you can write

```
str s = string !
#include "file"
!;
```

to get the file `file` included in the string `s`.

Note, that in EFEU (and so in esh) null strings (character pointer to NULL) can be used like ordinary strings and there is a difference between null strings and empty strings (strings containing only the terminating 0) are handled. The EFEU libraries contains tools for handling dynamic allocated strings and you can mix them with constant strings. The memory allocation tools in EFEU knows, if the memory of a string could be freed.

### Pointer types

Data types, which are implemented by pointers, assign only the pointer address and do not copy the data (strings are an exception). But they use a reference counter for garbage collection. If the pointer is no more used, the whole memory allocated to this pointer is freed. All this types are subclasses of the type `_Ref_`.

The type `_Ref_` and all other types with a pointer representation in C are subclasses of `_Ptr_`. This is also the type of the constant `NULL`.

Types which starts and ends with the underline symbol are reserved for internal use. Normally, you don't declare variables of this type. But this types may be used for arguments in virtual functions, e.g. to distinguish between the constant `NULL` and a string initialized with `NULL`.

### Lists

Lists are a ordered set of any data variables. They have the data type `List_t`. There are three ways to get lists in the EFEU interpreter language:

1. with the list grouping operator: `{ 3, 5 }`. You can not use it at begin of a statement because `{` is also used for grouping expressions.
2. with the list operator: `3, 5`. Note the low priority of the list operator. You need parentheses to use it in terms. You get only lists of at least two elements.
3. with the range operator: `3 : 5 : 2`. All members of the list have the same type.

Any object of type `List_t` have the two components:

`obj`

returns the first entry of the list or `NULL` for empty lists.

`next`

returns the sub list starting with the next element or an empty list, if there are no more elements.

In absence of pointers in esh, you may use `List_t` as substitution.

### Data fields

Data fields may be declared in one of the two possible forms:

`type name[dim]`

declares `name` as a field of type `type`.

`type[dim] name`

declares `name` as a scalar of type `type[dim]`.

In the first case and if the data field is initialized by a list of values, `dim` may be omitted and the number of elements in the list determines the size of the field. In the second case, a new type is implicit created. The field size is necessary.

In the second case 0 or a missing value of `dim` indicates a variable length array. The data field is implicit enlarged as you use a higher index. Data types of the form `type[]` are subclasses of `EfiVec`.

If you have more than one dimension, a declaration of the form

```
type name[n0][n1]...[nk];
```

is translated into

```
type[nk]...[n1 name][n0];
```

because data fields can only have one dimension, but there is no limit on creating vector types. It is clear that only *n0* may be omitted.

Data fields are always packed into a object of type `EfiVec` on use. A data field can always be converted to a list and you can assign values to a data field with a list. If the list has less elements than the data field, only the corresponding elements are changed.

EFEU provides you the following data types for a more powerful handling of data than ordinary data fields:

#### `TimeSeries`

are dynamic fields of type double with a time index.

#### `mdmat`

holds a data cube of any type with unlimited number of dimensions, only restricted by available memory. EFEU contains a lot of tools for handling such data cubes.

### Creating new data types

The simplest way to create a new type is `typedef`, as in

```
typedef int myint;
```

The new type `myint` is created as subclass of `int`, not as alias.

Structures are created with the `struct` statement. The syntax is

```
struct type [: base [ name ]] { type declarations }
```

like in C++. If *base* is defined, *type* is created as subtype of *base*. Only one base type may be specified.

The following two types

```
struct T1 {
    int a;
    int b;
}

struct T2 : int a {
    int b;
}
```

have the same components, but `T2` can be used as representation of an integer.

Any previously defined type can be used in this form of type declaration. Any structure type could be converted to a list and any list with corresponding elements could be converted to a structure type.

Example for a more complex structure:

```
struct MyDataType {
    int i;
```

```

    double d;
    str s;
    int v[10];
};

```

The EFEU interpreter supports enumeration types. The syntax is

```
enum type [: base [ name ]] { identifier list }
```

with a comma separated list of identifiers with optionally assignment values. If *base* is defined, *type* is created as subtype of *base*, else as subtype of `_Enum_`. The enumeration keys are bound to the enumeration type. They may be used immediately after declaration.

The following statement:

```
enum Color { Red, Green = 5, Blue };
```

creates an enumeration type with name `Color` and identifiers `Color::Red`, `Color::Green` and `Color::Blue`. The corresponding integer values are 0, 5 and 6. For every enumeration type, converts from/to `int/str` are created. So the following assignments are all valid:

```

Color c1 = "Red";
Color c2 = 0;
str s = Color::Red;
int n = Color::Red;

```

The function `enumlist(type)` returns a list of all valid identifiers of the enumeration type *type* or an empty list, if *type* has no identifiers or is not an enumeration type.

## FUNCTION DECLARATION

A function declaration in esh has the general form

```
type name ( arglist )
      expr
```

Normally *expr* is a block structure, but in esh you can also use a single (but not empty) expression. If the function does not return any value, use the special type `void`.

The following function declarations are equivalent:

```

int f (int x) x + 1;
int f (int x) return x + 1;
inline int f (int x) { return x + 1; }

```

In esh, the keyword `inline` has nothing to do with optimization, but with visibility. An inline function sees all variable tables as in the line where it is called. All functions defined with a single expression are default of type inline.

Here is an example where inline functions are needed:

```

inline str f (str fmt)
{
    return psub(fmt);
}

{
    str x = "foo";
    f("x = $(x)");
}

```

The function `psub` substitutes parameter according to a format string. If `f` is not inline, `psub` does not see the variable `x` and the substitution `$(x)` would fail.

Functions have the type `Func` and you can use it like function pointers in C. Typing the function name in outermost level gives you the prototype of the function.

As in C++ function arguments may have default values. The general form of a function argument is:

```
type [ & ] name [ = value ]
```

The `&` key indicates that the argument must be an lvalue. A tilde `...` stands for a variable list of arguments. Inside the function this list could be referenced under the reserved name `va_list`.

Behind the most operators stands a function with the name of the operator. You can use either `operator "name"` or `operatorname<space>` for such function names. For example: `operator+` is the name of the addition function. Function names of prefix operators have an additional `()` in the name to distinguish between postfix and binary operators. So `operator+()` is the name of the unary plus.

### Virtual functions

As in C++, you can overload functions with different argument lists. The keyword `virtual` is used to declare virtual functions. They have the data type `VirFunc`. Any function can be converted to a virtual function.

You can convert a virtual function to a regular function with a prototype cast:

```
Func f = operator+ (int a, int b);;
```

Now you can use `f` to add two integer values. Note the two semicolons: The first is part of the prototype, the second terminates the expression and may be replaced by a linefeed.

### Type bound functions

Functions may be bound to an specific type. They have the general form

```
type btype::name [ & ] ( arglist )
      expr
```

The `&` after the function name indicates that it can only be used for lvalues. A bounded function is called

```
obj.name(args)
```

where `obj` is an object of type `btype`. Object bound functions have the type `ObjFunc` and may be virtual or not.

All assignment operators are bounded functions. In bounded functions you can use `this` to refer to the corresponding object.

### Special Functions

Functions which have the same name as a type, defines constructors and converters.

Constructors have the form:

```
virtual type type ( arglist )
```

The special form

```
type type ()
```

is called the copy constructor.

The declaration

```
type type (void)
```

is a normal constructor without arguments.

Converters have the form:

```
tg_type src_type ()
```

with an empty argument list. The source data is referred under the name `this`. If the target type is `void` the function defines the destructor of the type.

Because of internal garbage collection, there is normally no need on copy constructor and destructor. You must be very carefully in defining this functions, because you get an infinite recursive call if an object of the type is copied inside the function.

## MISSING SOMETHING

If you are missing something in the documentation, you may get it from esh. The option `--info` provide an interface to builtin information, just enter the command `esh --info`. If you are running esh in interactive mode, you can get the information with the function call `Info()`.

If you want to know how a function is used, just enter the function name and the prototype is displayed. For a data type *type* you can call the function *type.info()* to get additional information's.

`global`

is the table of global variables

`local`

is the table of local variables. In outermost level `local` is identical to `global`.

`str whatis (.)`

give you information about the argument. returns the type of an variable.

`void vtabstack (int = 0, IO = iostd)`

show the current stack of the variable tables.

`List_t typelist ()`

give you a list of all data types.

## ENVIRONMENT

`APPLPATH`

path for configuration files.

`LANG`

locale information

`ESHPATH`

define extra directories for searching script headers.

## COPYRIGHT

Copyright (C) 1994, 2001 Erich Frühstück

**NAME**

md2edb – convert data cube to efeu data files

**SYNOPSIS**

```
md2edb [ --help[=type] ] [ --version ] [ --info[=entry] ] [ --debug[=mode] ] [ --verbose ] [ -1 ]
      [ --fast ] [ -9 ] [ --best ] [ -E expr ] [ -F file ] [ -v var ] [ -r ] [ -T title ] [ -p mode ] [ -d ]
      [ -n lim ] [ -f def ] in { name=var } [ out ]
```

**DESCRIPTION**

The following options and arguments are accepted from command `md2edb`:

- `--help[=type]`  
create command usage. The optional parameter *type* determines the formatting and output of the description.
- term  
display on terminal (default)
- raw  
output raw data for `efeudoc`
- man  
output nroff/troff source for `man`
- lp send postscript data to `lpr`
- `--version`  
show version information
- `--info[=entry]`  
show command information
- `--debug[=mode]`  
set debug level for command See for details.
- `--verbose`  
set debug level to `.info`.
- `-1, --fast`  
use fast compression by writing with `gzip`.
- `-9, --best`  
use best compression by writing with `gzip`.
- `-E expr`  
evaluate expression *expr*
- `-F file`  
execute command lines in file *file*
- `-v var`  
select type components of data cube
- `-r` remove singular axis
- `-T title`  
set title of output file
- `-p mode`  
output mode

- `-d` print data without header
- `-n lim`  
Limit of output data lines
- `-f def`  
apply filter *def* on efeu data files
- `in` input file name
- `name=var`  
selection parameter for data cube
- `out` output file name

## ENVIRONMENT

`APPLPATH`  
path for configuration files.

`LANG`  
locale information

## COPYRIGHT

Copyright (C) 2006 Erich Frühstück

# Bibliography

- [1] Erich Frühstück. The EFEU Interpreter EFEU-Documentation