SIEMENS

Table of Contents

1. Parser DLL
2. Functionality
3. Functions
    3.1. easy_parse
    3.2. write_element

EasyCODE

    3.3. write_level_start
    3.4. write_level_end

Parser Interface

    3.5. write_tree_start
    3.6. write_tree_end
    3.7. parse_error
    3.8. write_debug
    3.9. config_dialog
4. Error Messages
5. Debugging
6. Parser Configuration

Techn. SW Documentation

7. Constructs
    7.1. Notation
    7.2. List of Constructs
        1. DUMMY
        2. BS2
        3. IF
        4. WHILE
        5. CYCLE
        6. BREAK
        7. CASE
        8. CASEBRANCH
        9. AND
        10. OR
        11. NOT
        12. COND
        13. BLOCK
        14. LEVEL
        15. COMMENT
        16. SWITCH
        17. SWITCHBRANCH
        18. FOR
        19. REPEAT
        20. CALL
        21. WHEN
        22. EXIT
        23. DETACH
        24. LEAVE
        25. IFERROR
        26. AGBLOCK (Action block)
        27. JETPROC
        28. ISP (FREE FORMAT)
        29. C_SWITCH

EasyCODE Version 6.xE, 06-20-1996

        30. C_CASE
        31. DEFAULT

32. RETURN
33. VARIABLE
34. COB_PROGRAMM
35. COB_SECTION
36. COB_PARAGRAPH
37. COB_INLINE (Inline Perform)
38. COB_TIMES (Perform Times)
39. COB_VARYINGAFTER (Perform varying after)
40. COB_EXITPER (Exit perform)
41. COB_EXITTEST
42. COB_EXITPROG (Exit program)
43. COB_CALL
44. COB_EXCEPTION
45. COB_EVALUATE
46. COB_SEARCH
47. ENTRY
48. PROC
49. AUSWAHL
50. WIEDER
51. RAHMEN
52. PET_BLOCK
53. PET_AGBLOCK
54. PET_JUMPRESTART
55. PET_FOR
56. PET_WHILE
57. PET_REPEAT
58. PET_IF
59. PET_ELSE
60. PET_IFCMDERROR
61. PET_IFBLOCKERROR
62. XOR
63. PET_PROC
64. PET_IFBRANCH
65. CLASS
66. PRIVATE
67. PUBLIC
68. PROTECTED
69. PROG_AUFRUF
70. BTMITTEL
71. VARIANTE
72. ASS_THRU
73. ASS_CYCLE
74. ASS_CASE
75. ASS_EXIT
76. PET_INTPROC
77. FUNC

8. Use of Constructs

SPX
DS
COB
CPP

9. File-Specific Information

9.1. Short Info
9.2. IF Layout
9.3. Segment Numbers
9.4. Line Numbers
9.5. Screen Font
9.6. Printer Font
9.7. Highest Level-ID

10. Examples

Example 1

Source...
Function calls...
Structure Diagram..
Example 2...
Source...
Function calls...
Structure Diagram..
Example 3...
Source...
Function calls...
Structure Diagram..
Index...

1 Parser DLL

EasyCODE V3.51 and higher versions allow you to have existing source code analyzed by a DLL. This parser DLL is used when a file which does not have EasyCODE format is selected to be opened in EasyCODE. This means that source code or any other file which does not have EasyCODE format can be processed by the parser DLL.

The parser DLL is used for reading the file (from now on referred to as source) and telling EasyCODE which constructs are contained in the source. To communicate with EasyCODE, the parser DLL uses function calls containing as parameters the construct type and the text contained in the constructs. On the basis of this information, EasyCODE will build up an internal representation of the source and display it on the screen.

In EasyCODE V3.51, the parser interface is available in the components DS, SPX and COB. In version V4.0 and higher versions, this interface is available in all components and used for the import function.

The name of the parser DLL is specified in the INI file (SPX: in the configuration file):

```
[Settings]
Parser=parser.dll
```

If this entry does not exist, the default names EASY-BNF.DLL and EASY-PAR.DLL will be used for DS and for COB, respectively.
In SPX, the entry is added to the configuration file in the following section:

```
[ParseOptions]
Parser=parser.dll
```

In SPX, there is no default value. If the entry does not exist, the parser will not be called, and the source will be rejected as not having EasyCODE format.
EasyCODE will search for the DLL in the module directory. You may, however, also enter a full path name.

2 Functionality

The parser DLL is activated by the point of entry easy_parse. A pointer to a structure containing further information is specified as a parameter. This information includes a reference to the source file to be analyzed as well as pointers to callback functions which can be called by the parser.

The source file is read by the parser. If the latter detects constructs supported by EasyCODE (see chapter Constructs), then callback functions will be called. By calling the callback functions, the parser creates a file in an internal format which EasyCODE is able to display on the screen. The parser must be able to interpret source files coded in compliance with both DOS and Unix conventions (CR/LF or just LF; ^Z optional); at any rate, the DOS convention for generating line breaks (CR/LF) must be used for all strings delivered at the parser interface. This means that the parser must expand single LFs to CR/LF combinations, if required.

The following callback functions are provided for writing the internal file format[1]:

```
write_tree_start
write_tree_end
write_element
write_level_start
write_level_end
```

The calling sequence of the callback functions must comply with certain conventions:
The entire structure diagram consists of a sequence of constructs. This sequence is called TREE. A TREE begins with write_tree_start and ends with write_tree_end.

Every construct is introduced by write_element. Since a construct may contain other constructs, write_element is followed by a certain number of TREEs (depending on the construct)..

A special feature of the construct is the segment (level). A segment begins with write_level_start and ends with write_level_end. The contents of the segment is enclosed between these two entries as a sequence of constructs (TREE). If the segment is empty, write_level_start is immediately followed by write_level_end.
Text between the keywords of the constructs is delivered as a comment construct[2].
Text sent to EasyCODE must not contain any tabs[3]. Every text must be ANSI coded. Line breaks are indicated by \r\n. The maximum number of characters within text including the terminating \0 must not exceed MAX_TEXT_SIZE characters.
The following grammar[4] conventions apply:

```
FILE ::= write_tree_start LEVEL write_tree_end

LEVEL ::= write_level_start TREE write_level_end

LEVEL ::= write_level_start write_level_end

TREE ::= write_tree_start (ELEMENT)+ write_tree_end

ELEMENT ::= LEVEL

ELEMENT ::= write_element (TREE)*
```

The number of TREEs following write_element depends on elementTyp.

The entire file consists of a TREE containing one single construct, a LEVEL, which is referred to as the top level. The level comment of the top level is displayed in the header of the structure diagram.
A simple example:

Source:

```
A:=0;
```

[1] This internal file format is represented by a tree, therefore the functions are named write_tree_start etc.
[2] A more suitable name for this construct would be text construct.
[3] Therefore the parser DLL must replace existing TABs with the corresponding number of blanks.
[4] * means 0 or more repetitions, + means 1 or more repetitions.

```
     B:=5;
```

Calls: (sequences of pseudocode parameters are not complete!)

```
     write_tree_start(node_level_typ, edge_rumpf,...)
     write_level_start(...)                                TOP LEVEL
     write_tree_start(level_typ, edge_rumpf, ...)
     write_element(node_comment_typ, "A:=0")               COMMENT
     write_element(node_comment_typ, "B:=5")               COMMENT
     write_tree_end()
     write_level_end(...)
     write_tree_end()                                      TOP LEVEL ENDE
```

3 Functions

For definitions concerning the interface, see C-Include File 'parse.h'.

4 easy_parse

Purpose:

The parser DLL provides the easy_parse function which is called by EasyCODE. The name of the easy_parse function must be exported by the parser DLL.

Definition:

```
int FAR PASCAL easy_parse
                    (
                    LP_PARSER_DATA parser_data
                    );
```

Parameters:

parser_data ... The interface data are delivered in a structure. parser_data is a FAR Pointer to this structure.

```
typedef struct
{
  int interfaceVersion;
  HWND hWnd;
  char komponente [4];
  int hSourceFile;
  LPBUFFILE hFile;
  char envFileDir [_MAX_PATH];
  char configFileName [_MAX_PATH];
  LP_WRITE_ELEMENT        lp_write_element;
  LP_WRITE_LEVEL_START_V2 lp_write_level_start;
  LP_WRITE_LEVEL_END      lp_write_level_end;
  LP_WRITE_TREE_START     lp_write_tree_start;
  LP_WRITE_TREE_END       lp_write_tree_end;
  LP_PARSE_ERROR          lp_parse_error;
  LP_WRITE_DEBUG          lp_write_debug;
  LPSTR lp_stack_bottom;
  BOOL FAR *bECSource;
  unsigned FAR *lpNumErrors;
  unsigned FAR *lpNumWarnings;
  char spxConfigFileName [_MAX_PATH];        (EasyCODE V4.0 + higher,
                                             PARSE_VERSION >=2)
  BOOL bOemSource;                           (EasyCODE V4.0 + higher
                                             PARSE_VERSION >=3
} PARSER_DATA;

typedef PARSER_DATA FAR * LP_PARSER_DATA;
```

interfaceVersion ... Version of the interface. The current version number is defined by the PARSE_VERSION macro in parse.h. With each modification of the interface, the version number will be increased by 1, so that a parser will be able to find out whether it is called by an EasyCODE version supporting an older or newer version of the interface.

hWnd ... The Window handle of the calling window.

komponente ... Component identifier of the calling EasyCODE application. The string consists of a maximum of 3 characters. ('SP', 'SPX', ‘C’, 'CPP', 'COB', 'DS'). This ensures that the parser will be called by a suitable component only. The COB parser, for example, may not be used for SPX.

hSourceFile ... DOS handle of the source file. Before the file is delivered, it is opened with O_BINARY. The run time function read() (or _read in MSC7 or lread from the SDK - we recommend the one from the SDK) should be used for reading the file. After the parser run, the file must not be closed.

hFile ... Handle of the file in internal EasyCODE format. This is a pointer to an internal EasyCODE structure, which is used for buffering write accesses. This is the reference given to the interface functions for writing the EasyCODE format. This file handle may not be used by the parser for direct reading or writing.

envFileDir ... Full path name of the EasyCODE module directory. This path name may be used for loading modules for the parser or for finding files required by the parser (e.g. file for error messages). The path name ends with '\'..

configFileName ... Pathname of a file which may be used for reading data concerning the parser configuration. In SPX, this is the SPX configuration file, in all other components, the INI file.
In version V4.0 of EasyCODE and V2 of the interface or higher versions, this field contains the name of the INI file even in SPX. The name of the SPX configuration file is specified in the spxConfigFileName field.

lp_write_element,

lp_write_level_start,

lp_write_level_end,

lp_write_tree_start,

lp_write_tree_end,

lp_parse_error

lp_write_debug ... FAR pointer to callback functions used for writing the EasyCODE file format and for communicating with the user. The callback functions are provided by EasyCODE. For a description of the functions see below.

lp_stack_bottom ... FAR pointer to the address of the pseudo variable 'end' from the C run time system. The pointer marks the end of the data area and the beginning of the stack area in the EasyCODE data segment. Since the DLL also uses the EasyCODE stack, this pointer may be used for avoiding a stack overflow of the parser. The stack available to the parser is about 20K.

bECSource ... Return parameter. Pointer to a variable which must be TRUE, if an EC-generated source is read. The variable must be set by the parser, as soon as this fact is known. It may even be used in the EasyCODE callback functions (e.g. parse_error()) . Not available in the current version 4.0.

lpNumErrors ... Return parameter. Pointer to a variable in which the parser must return the number of errors that occurred during file analysis.

lpNumWarnings ... Return parameter. Pointer to a variable in which the parser must return the number of warnings that occurred during file analysis. A problem detected by the parser during file analysis must either be classified as a warning or as an error. The message must be delivered to the user via the parse_error callback function.

spxConfigFileName ... Available in EasyCODE V4.0 and interfaceVersion 2 and higher versions. In SPX, this field contains the name of the SPX configuration file, in all other components, it contains the empty string.

bOemSource ... Available in EasyCODE V4.0 and interfaceVersion 3 and higher versions. Determines whether the source file complies with the OEM character set. If this parameter is TRUE, the parser must convert text to the ANSI character set and deliver ANSI text to EasyCODE.
If this field does not yet exist, the value must be assumed TRUE.

Return value:

A value defined by the following macros:

PARSE_OK ... If the analyzing procedure was successfully completed. No warnings or errors occurred.

PARSE_WARN ... Only warnings, but no errors occurred. In this case, the structure diagram will be displayed. If the callback function calls did not result in creating a correct and complete structure diagram, this return value must not be specified, since the structure diagram could not be displayed correctly.

PARSE_ERRORS ... Errors occurred. The structure diagram will not be displayed.

PARSE_STACK ... The parser stopped because of stack overflow. The warning comes from EasyCODE.

PARSE_REENTER... The parser DLL has already been called. If the DLL is not reentrant[5], it can reject any further call with this return value. The warning comes from EasyCODE.

PARSE_MEMORY... Lack of memory during parser execution. The warning comes from EasyCODE.

Callback functions are provided by EasyCODE.

5 write_element

Purpose:

To write a construct.

Definition:

```
int FAR PASCAL write_element
                   (
                   LPBUFFILE hFile,
                   int zeile,
                   enum parse_node_typ elementTyp,
                   COBSTRING string1,
                   COBSTRING string2,
                   COBSTRING string3,
                   int par1,
                   int par2
                   );
```

[5] In Windows, this is the case if the DLL uses global variables (in their own data segment) and takes no further steps (such as assigning global variables to calling programs or the like), since in Windows, a DLL will always be loaded only once. In Windows NT, this problem has been eliminated (DLLs may be DATA MULTIPLE).

Parameters:

hFile	...	The hFile delivered when easy_parse is called
zeile	...	Line number to be assigned to the construct
77elementTyp	...	Type of the construct.
string1 .. 3	...	Used for delivering the required text. The meaning of these strings depends on the construct type. Parameters that are not required must be set to 0 or text=NULL.
par1 .. 2	...	Used for delivering numerical information. The meaning of these strings depends on the construct type. Parameters that are not required must be set to 0.

Return value:

0	Ok
-1	Error. Analysis should be cancelled.

COBSTRING is a structure used for delivering strings.

```
typedef struct
{
LPSTR text;
int zeile;
int spalte;
} COBSTRING;
```

text	...	Delivered text. The text may consist of several lines, with the line breaks indicated by '\r\n'. Text delivered to EasyCODE must not contain any tabs. Every text must be ANSI coded. The text including the terminating \0 must not exceed a maximum of MAX_TEXT_SIZE characters.
zeile	...	Line number to be assigned to the first line in the text.
spalte	...	Column number to be assigned to the first character in the text. Every new line will begin in column 1.

The neutral form of the Cobstring (NULL-Cobstring) with text = NULL, zeile = spalte = 0 is used in case the Cobstring parameter is not necessary.

6 write_level_start

Purpose:

Beginning of a new level (segment). Levels allow you to divide a structure diagram into segments, so that some elements will not be visible. These elements will be displayed as separate structure diagrams. This results in a hierarchical structure of the structure diagram.

This hierarchical structure is absolutely necessary for large structure diagrams, because otherwise, the maximum size[6] of a segment[7] would be exceeded. Therefore, the source code should automatically and in a reasonable way be divided into segments during file analysis. Every function may, for instance, become a separate segment. The function name may be used as a segment comment. If the parser is able to distinguish between sources generated by EasyCODE and "native" sources, only "native" sources should be pushed down automatically, while the segment information generated by EasyCODE should be observed when it comes to EasyCODE sources.

Every write_level_start call requires a corresponding write_level_end call, which ends the segment.

[6] The maximum size of a segment depends on the resolution of the output device and the selected font. The structure diagram to be displayed may comprise about 32000 pixels in width and height.
[7] The top level of a structure diagram is also displayed as a segment.

Definition:

```
    int FAR PASCAL write_level_start
                        (
                        LPBUFFILE hFile,
                        int zeile,
                        COBSTRING ebenenKommentar,
                        LONG FAR * posOfLevel,
                        DWORD levelID,
                        COBSTRING entryName
                        );
```

Parameters:

hFile	...	The hFile delivered when easy_parse is called
zeile	...	Line number to be assigned to the segment
ebenenKommentar	...	Text for the segment comment[8]. If no comment exists, a NULL-Cobstring must be delivered. Within the *zeile* structure element, the line number from the source file of the first line of the comment or level must be specified.
posOfLevel	...	Return parameter. The value of this variable is delivered by EasyCODE and must be stored for the corresponding WRITE_LEVEL_END call.
levelID	...	ID of the level. Level-IDs are used when OLE is concerned. If the level-IDs are to be delivered by the parser, all level-IDs must differ from each other in order to ensure that EasyCODE will run correctly. The maximum of all level-IDs must be delivered as the file-specific information 'node_lastlevelid'. If there are no level-IDs in the source or if they are not analyzed by the parser, then this parameter must be set to 0 for all calls. In this case, the file-specific information 'node_lastlevelid' must not be delivered.
entryName	...	Is used for COL only and specifies the name of the entry indicating the segment. In all other components, this parameter must be a NULL-Cobstring.

Return value:

0	Ok
-1	Error. Analysis should be cancelled.

[8] Makeshift solution for import DLL: In the case of EasyCODE(COB), the Offset for the level comment will be delivered in the component column of the COBSTRING. This should be changed in case a new COBOL parser should become available.

7 write_level_end

Purpose:

Indicates the end of a segment.

Definition:

```
int FAR PASCAL write_level_end
                    (
                    LPBUFFILE hFile,
                    LONG posOfLevel,
                    LONG posSource,
                    LONG lengthSource,
                    int zeilen
                    );
```

Parameters:

hFile	...	The hFile delivered when easy_parse is called
posOfLevel	...	The value corresponding to the write_level_start call is delivered here.
posSource	...	Position of the beginning of the segment in the source. This parameter must be set to 0.
lengthSource	...	Length of the segment in the source. This parameter must be set to 0.
zeilen	...	Number of lines contained in the segment in the source. This parameter must be set to 0.

Return value:

0	Ok
-1	Error. Analysis should be cancelled.

8 write_tree_start

Purpose:

Indicates the beginning of a TREE. Every write_tree_start call requires a corresponding write_tree_end call which ends the TREE.

Definition:

```
int FAR PASCAL write_tree_start
                    (
                    LPBUFFILE hFile,
                    enum parse_node_typ predTyp,
                    enum parse_edge_typ edge
                    );
```

Parameters:

hFile	...	The hFile delivered when easy_parse is called.
predTyp	...	Type of the construct containing the TREE.
edge	...	Edge of predTyp containing the TREE.

Return value:

0	Ok
-1	Error. Analysis should be cancelled.

Comment:

The predTyp and edge parameters are dealt with in a similar way as in the DUMMY construct. The values to be applied are specified there. For the top TREE, the values node_level_typ and edge_rumpf are specified.

9 write_tree_end

Purpose:

Indicates the end of a TREE.

Definition:

```
int FAR write_tree_end
                        (
                        LPBUFFILE hFile
                        );
```

Parameters:

hFile	...	The hFile delivered when easy_parse is called

Return value:

0	Ok
-1	Error. Analysis should be cancelled.

10 parse_error

Purpose:

Output of an error message. Error messages or warnings occurring during file analysis will be delivered to EasyCODE with this function.

Definition:

```
typedef int FAR PASCAL parse_error
                        (
                        int zeile,
                        LPSTR fehlerText,
                        BOOL wiederAufsetzen
                        );
```

Parameters:

zeile	...	Line number in the source in which the error occurred
fehlerText	...	Text of the error, ANSI coded, line breaks with "\r\n" are permitted. At the end of the text, there should be no line break. The text must be terminated with '\0', its lenght is restricted to 64K.
wiederAufsetzen	...	TRUE, if the analysis may be continued

Return value:

0	continue analysis
-1	cancel analysis

Comment:

The error text must contain all necessary information, even the line number, in the form of text. The error text is written to a file and remains unchanged.

If user interaction is possible when the text is displayed to the user (this is not the case in EasyCODE V4.0), and if the user wants to cancel the parser run, the function will return with the return value -1.

11 write_debug

Purpose:

To write debug text into the debug file.

Definition:

```
int FAR PASCAL write_debug
                   (
                   LPSTR debugText
                   );
```

Parameters:

debugText ... Text which is to be written. The text is written into a debug file together with internal EasyCODE debug text. The text must be ANSI coded and may contain several lines. Its length is restricted to 64K. After the text, a line break '\r\n' will be written into the debug file.

Return value:

0	continue analysis
-1	cancel analysis

Comment:

For details on the debug file, see chapter "Debugging".

12 config_dialog

Purpose:

The parser DLL provides the config_dialog function, which can be called by EasyCODE.

The function should open a dialog window, in which parser-specific options may be configurated.

Definition:

```
int FAR PASCAL CONFIG_DIALOG
                        (
                        HWND hWnd,
                        LPSTR envFileDir,
                        LPSTR configFileName,
                        LPSTR spxConfigFileName
                        );
```

Parameters:

hWnd ... The Window handle of the calling EasyCODE window

envFileDir ... Full pathname of the EasyCODE module directory. This path name may be used for loading modules for the parser or for finding files required by the parser to ensure its correct function (e.g.: file for error messages). The path name ends with '\'.
The path name is identical with the one delivered with easy_parse.

configFileName ... see easy_parse

spxConfigFileName ... see easy_parse (V4.0 only)

Return value:

not defined

Comment:

The name of the function ("config_dialog") must be exported by the parser DLL.

If no application exists for this function, the name must not be exported.

For details see chapter "Parser Configuration".

13Error Messages

Error messages and warnings are delivered to EasyCODE with the parse_error callback function. The delivered text should contain the line number, the type of the error and the error text, e.g.:

```
(561): Error: 'else' without 'if'
(565): Warning: Unexpected EasyCODE comment
```

The error text is written to an error file by EasyCODE, and a line break is added. Internally, the parser must count the number of errors and warnings that occurred and return them as return parameters lpNumErrors and lpNumWarnings. If at the end of the parser run one of the return parameters is greater than 0 and an error file exists, this error file will be displayed by EasyCODE in an editor. According to the number of errors and warnings, the return value of the parser must be specified. If the return value is PARSE_OK or PARSE_WARN, the structure diagram will be displayed.

Messages must be classified as warnings or as errors in such a way that warnings will not affect the graphic representation of the structure diagram. If a correct display of the source code in the structure diagram cannot be ensured or if the parser did not complete its run, then PARSE_ERROR must be returned as the return value of easy_parse.

14Debugging

By adding an entry to the INI file (SPX: configuration file), you may specify a file to which debug text is to be written during the parser run.
For SPX in the configuration file:

```
[ParseOptions]
ParserDebugFile=<filename>
```

For all other components in the INI file:

```
[Settings]
ParserDebugFile=<filename>
```

The EasyCODE callback functions add corresponding entries to this file specifying the name of the callback function and parameters. You may write your own text by calling the parse_debug function.
If no ParserDebugFile has been defined or if it cannot be opened, parse_debug calls will be ignored.

15Parser Configuration

With the config_dialog function, the parser DLL may provide another point of entry allowing parser configuration. Parser configuration can be necessary for switching on or off certain syntax extensions or dialects by way of options, in the same way as in a compiler.

EasyCODE calls the config_dialog function, when the user selects it with the help of the EasyCODE user interface. The function should then load the parser options from the file named configFileName, display it for modification in a dialog window and then save it again in the file. The parser DLL cannot assume that it will remain loaded until the easy_parse function is called. Therefore the configuration data cannot be stored until the parser is actually called. Before the parser run, the configuration data must also be loaded from configFileName.

The appearance of the dialog box, the format, in which the configuration data are stored in configFileName and the type of the configuration data depend on the parser DLL and can therefore not be defined here. At any rate, this information should be written to a separate parser-specific section, the name of which is designed as follows:

```
[<filename>.<ext> <ver>]
```

<filename> indicates the base name, <text> the extension of the parser filename and <ver> an internal parser version number for interpreting the entries, e.g.

```
[CLIPPER.DLL V1]
```

If there is no way of configurating a parser DLL, the parser DLL should not export a function named config_dialog. The EasyCODE user interface will not provide a way of configurating the parser.
In EasyCODE version V3.51, there is no way of calling this point of entry in EasyCODE.
In version V4.0, this possibility is not implemented.

16Constructs

17 Notation

<numerical value of construct type>. name of construct
elementTyp = <construct type>
par1 = <symbolic name> (type)
string1 = <symbolic name>
TREE <symbolic name> <edge type>

For every construct, write_element is called with the construct type as a parameter.

The parameters par1-2 and string1-3 required for the write_element function are specified for every construct. The parameters are assigned symbolic names which will then be used for the description. Parameters that have not been specified must be set to 0.

The parameters are followed by a number of TREEs. Each tree is followed by the corresponding edge type which must be specified in case of write_tree_start or a dummy in the TREE.

Please note that the following edge types require certain constructs:

edge_text	Must be followed by a single comment or a dummy.
edge_bedingung	Must be followed by a condition construct (AND, OR, NOT, XOR), a list of comments or a dummy.

If these requirements are not fulfilled, this may result in structure diagrams which cannot be created by EasyCODE and which may show unpredictable behavior when edited with EasyCODE.

18 List of Constructs

1. DUMMY

elementTyp = node_dummy_typ

par1 = pred_typ (enum parse_node_typ)
par2 = follows_as (enum parse_edge_typ)

Parameters:

pred_typ	...	Construct type of the construct on a higher level
follows_as	...	Edge of the construct on a higher level, to which the dummy construct belongs.

Comment:

The dummy construct is a special construct. It is always used when the TREE would otherwise be empty, because there is no corresponding entry in the source. If the ELSE branch of an IF is empty, the TREE for the edge edge_else must contain exactly one dummy construct including the parameters par1=node_if_typ and par2=edge_else.

Between the write_tree_start and write_tree_end calls, there must always be a write_element call.

2. BS2

elementTyp = node_bs2_typ

par1 = bs2_typ (enum bs2_typ)

string1 = info_string

Parameters:

par1 ... Specifies the subtype of the action:

sdfcommand_typ	...	SDF Command
sdfstatement_typ	...	SDF Statement

info_string ... The entire information concerning the action is stored by the parser in an info string. This string is analyzed in detail by EasyCODE. For this purpose, the parser interface calls the InfoToJet function, which creates a JET structure from the basic type specified by par1 and the info string. The JET structure will then be written to the TMP file and released again by the parser interface.

For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

Since the info string is checked by the application and not by the parser DLL, error messages will also come from the application. To allow the application to display error messages containing reasonable information about the exact location of the invalid info string within the ETF file, the line number of the first line of the info string must be specified in the *zeile* structure element of the info string.

3. IF

elementTyp = node_if_typ

string1 = label (COL only)

TREE Condition	edge_bedingung
TREE Then branch	edge_then
TREE Else branch	edge_else

Parameters:

label ... Labels may occur in several constructs. They are mainly required for COL and may otherwise be set to a NULL-Cobstring.

4. WHILE

elementTyp = node_while_typ

string1 = label (COL only)

TREE Condition edge_bedingung

TREE Body edge_rumpf

5. CYCLE

elementTyp = node_cycle_typ

string1 = label (COL only)

TREE Body edge_rumpf

6. BREAK

elementTyp = node_break_typ

TREE Condition edge_bedingung

7. CASE

elementTyp = node_case_typ

string1 = label (COL only)

TREE	Branch list	edge_zweigliste
TREE	Ofrest	edge_ofrest

Comment:

analogous to SWITCH/SWITCHBRANCH, but with CASE/CASEBRANCH.

8. CASEBRANCH

elementTyp = node_casebranch_typ

string1 = label (COL only)

TREE	Condition	edge_bedingung
TREE	Alternative	edge_alternative

9. AND

elementTyp = node_and_typ

TREE	Operands	edge_bedingung

Comment:

The logical operands AND, OR, XOR (not NOT) may have several digits, so that for instance an AND may have three operands. The three operands will then be arranged sequentially in the following TREE.

10. OR

elementTyp = node_or_typ

TREE	Operands	edge_bedingung

Comment:

See AND.

11. NOT

elementTyp = node_not_typ

TREE	Operand	edge_bedingung

Comment:

NOT may have only one operand. The TREE may therefore contain only one construct.

12. COND

node_cond_typ

string1 = info_string

Parameters:

info_string	...	The entire information concerning Cond is stored by the parser in an info string. This string will then be analyzed in detail by EasyCODE. See also BS2. For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

13. BLOCK

elementTyp = node_block_typ

string1 = label (COL only)

TREE	Header	edge_text
TREE	Body	edge_rumpf

14. LEVEL

elementTyp = node_level_typ

write_level_start(...)
TREE edge_rumpf
write_level_end(...)

Comment:

This construct represents a segment. It must not be written in combination with write_element, but the write_level_start and write_level_end functions must be called. Between these two functions, there will be the body of the level.

15. COMMENT

node_comment_typ

par1 = offset	(int)	(COB only, otherwise 0)
par2 = typ	(enum anweisungs_type)	(COB only, otherwise 0)
string1 = text		

Parameter:

offset ... Specifies the number of blanks that have to be inserted at the beginning of each line of text, so that the column position will be correct.

typ ... Specifies whether the construct will be used as an ordinary text construct (id_comment) or as a statement before (pre_division) or within a COBOL-Division (id_division, env_division, data_division, proc_division).

Comment:

This construct contains general text. It is used for statements, conditions and most other types of text. Several lines may be combined in the text. The maximum size of MAX_TEXT_SIZE characters must, however, not be exceeded.

If the construct is used for statements, the text may be divided into several statements in order to avoid maximum size or to structure text more clearly. The text may not be divided when the edge types edge_text, edge_klausel, edge_param, edge_anweisung, edge_max, edge_fuss or edge_bedingung (in those components that do not support AND/OR/NOT) are concerned, since in these TREEs only one comment construct is allowed. If the text exceeds the maximum size, an error message will appear, because the structure diagram cannot be displayed..

The offset and typ parameters will be used in COB only. In all other components, they must be set to 0.

16. SWITCH

elementTyp = node_switch_typ

string1 = label (COL only)

TREE	Variable	edge_text
TREE	Branch list	edge_zweigliste
TREE	Ofrest	edge_ofrest

Comment:

The branch list TREE must contain a sequence of Switchbranch constructs.

The Ofrest TREE must contain exactly one Switchbranch construct. If the value TREE of this Switchbranch construct will not be used (e.g. in EasyCODE(SP)), a dummy must be inserted into this value TREE.

17. SWITCHBRANCH

elementTyp = node_switchbranch_typ

string1 = label (COL only)

TREE	Value	edge_text
TREE	Alternative	edge_alternative

18. FOR

elementTyp = node_for_typ

TREE	Expression	edge_text
TREE	Body	edge_rumpf

19. REPEAT

elementTyp = node_repeat_typ

for COB:

TREE	Condition	edge_bedingung
TREE	Body	edge_rumpf

other components:

TREE	Body	edge_rumpf
TREE	Condition	edge_bedingung

Comment:

In the COB component, the Condition and Body TREEs have been exchanged.

20. CALL

elementTyp = node_call_typ

TREE	Call	edge_text

21. WHEN

elementTyp = node_when_typ

TREE	Condition	edge_bedingung
TREE	Label	edge_text

22. EXIT

elementTyp = node_exit_typ

Comment:

This construct does not require any parameters or subTREEs.

23. DETACH

elementTyp = node_detach_typ

Comment:

This construct does not require any parameters or subTREEs.

24. LEAVE

elementTyp = node_leave_typ

TREE	Label	edge_text

25. IFERROR

elementTyp = node_iferror_typ

TREE	Then branch	edge_then
TREE	Else branch	edge_else

26. AGBLOCK (Action block)

elementTyp = node_agblock_typ

TREE	Header	edge_text
TREE	Body	edge_rumpf
TREE	Abnormal	edge_abnorm

27. JETPROC

elementTyp = node_jetproc_typ

string1 = info_string

TREE	Header	edge_kopf
TREE	Body	edge_rumpf
TREE	Abnormal	edge_abnorm

Parameters:

info_string ... The entire information concerning Jetproc is stored in an info string by the parser. This string will then be analyzed in detail by EasyCODE.
See also BS2.

For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

28. ISP (FREE FORMAT)

elementTyp = node_isp_typ

string1 = text

Parameters:

text ... existing text

29. C_SWITCH

elementTyp = node_c_switch_typ

TREE	Expression	edge_text
TREE	Branch list	edge_zweigliste

Comment:

The TREE branch list must contain a sequence of no or up to several Switchbranch constructs and no or one default construct.

30. C_CASE

elementTyp = node_c_case_typ

TREE	Value	edge_text
TREE	Alternative	edge_alternative

31. DEFAULT

elementTyp = node_default_typ

TREE	Alternative	edge_alternative

32. RETURN

elementTyp = node_return_typ

TREE	Expression	edge_text

33. VARIABLE

elementTyp = node_variable_typ

string1 = info_string

TREE	Variant	edge_rumpf

Parameters:

info_string ... The entire information concerning VARIABLE is stored by the parser in an info string. This string will be analyzed in detail by EasyCODE.
See also BS2.

For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

34. COB_PROGRAMM

elementTyp = node_cob_programm_typ

TREE	Id	edge_kopf
TREE	Env	edge_env
TREE	Data	edge_data
TREE	Param	edge_param
TREE	Body	edge_rumpf

Comment:

This construct is used for the frame of a COBOL program in COB. Param contains the parameters after USING.

35. COB_SECTION

elementTyp = node_cob_section_typ

TREE	Name	edge_text
TREE	Bdoy	edge_rumpf

36. COB_PARAGRAPH

elementTyp = node_cob_paragraph_typ

TREE	Name	edge_text
TREE	Body	edge_rumpf

37. COB_INLINE (Inline Perform)

elementTyp = node_cob_inline_typ

TREE	Body	edge_rumpf

38. COB_TIMES (Perform Times)

elementTyp = node_cob_times_typ

TREE	Expression	edge_text
TREE	Body	edge_rumpf

39. COB_VARYINGAFTER (Perform varying after)

elementTyp = node_cob_varyingafter_typ

TREE	Expression	edge_text
TREE	Body	edge_rumpf

40. COB_EXITPER (Exit perform)

elementTyp = node_cob_exitper_typ

Comment:

This construct does not require any parameters or subTREEs.

41. COB_EXITTEST

elementTyp = node_cob_exittest_typ

Comment:

This construct does not require any parameters or subTREEs.

42. COB_EXITPROG (Exit program)

elementTyp = node_exitprog_typ

Comment:

This construct does not require any parameters or subTREEs.

43. COB_CALL

elementTyp = node_cob_call_typ

TREE	Call	edge_text
TREE	Parameter	edge_param

44. COB_EXCEPTION

elementTyp = node_cob_exception_typ

TREE	Clause	edge_klausel
TREE	Statement	edge_anweisung
TREE	Then branch	edge_then
TREE	Else branch	edge_else

Comment:

This construct is used for Exceptions in COB. The statement TREE may contain one single statement in the form of a comment construct or one single Cob_Call construct.

The clause TREE may contain only one single comment construct.

45. COB_EVALUATE

elementTyp = node_cob_evaluate_typ

TREE	Expression	edge_text
TREE	When list	edge_zweigliste
TREE	Other	edge_rumpf

Comment:

The When list TREE must contain a sequence of one or more Switchbranch constructs.
The Other TREE contains the TREE for the OTHER branch.

46. COB_SEARCH

elementTyp = node_cob_search_typ

TREE	Table	edge_text
TREE	At-End	edge_rumpf
TREE	When list	edge_zweigliste

Comment:

The When list TREE must contain a sequence of one or several Casebranch constructs.
The At-End TREE contains the TREE for the At-End branch.

47. ENTRY

elementTyp = node_entry_typ

TREE	Name	edge_text
TREE	Parameter	edge_param

48. PROC

elementTyp = node_proc_typ

TREE	Header	edge_text
TREE	Body	edge_rumpf

49. AUSWAHL

elementTyp = node_auswahl_typ

TREE	Branch list	edge_zweigliste

Comment:

The Branch list TREE must contain a sequence of one or several default constructs.

50. WIEDER

elementTyp = node_wieder_typ

TREE	Min	edge_text
TREE	Body	edge_rumpf
TREE	Max	edge_max

Comment:

The Max TREE as well as the Min TREE may contain exactly one comment or dummy construct.

51. RAHMEN

elementTyp = node_rahmen_typ

TREE	Header	edge_text
TREE	Body	edge_rumpf
TREE	Footer	edge_fuss

Comment:

The Footer TREE as well as the Header TREE may contain exactly one comment or dummy construct.

52. PET_BLOCK

elementTyp = node_pet_block_typ

string1 = label

TREE	Clause	edge_klausel
TREE	Body	edge_rumpf

53. PET_AGBLOCK

elementTyp = node_pet_agblock_typ

string1 = label

TREE	Clause	edge_klausel
TREE	Header	edge_text
TREE	Body	edge_rumpf
TREE	Abnormal	edge_abnorm

54. PET_JUMPRESTART

elementTyp = node_pet_jumprestart_typ

Comment:

No parameters, no subtrees.

55. PET_FOR

elementTyp = node_pet_for_typ

string1 = label

TREE	Expression	edge_isp
TREE	Body	edge_rumpf

56. PET_WHILE

elementTyp = node_pet_while_typ

string1 = label

TREE	Condition	edge_bedingung
TREE	Body	edge_rumpf

57. PET_REPEAT

elementTyp = node_pet_repeat_typ

string1 = label

TREE	Body	edge_rumpf
TREE	Condition	edge_bedingung

58. PET_IF

elementTyp = node_pet_if_typ

string1 = label

TREE	Branch list	edge_zweigliste

Comment:

The Branch list TREE must contain a sequence of 1 to n Pet_Ifbranch and 0 to 1 Pet_Else constructs, with a Pet_Else being the last construct in the sequence.

59. PET_ELSE

elementTyp = node_pet_else_typ

TREE	Alternative	edge_alternative

60. PET_IFCMDERROR

elementTyp = node_pet_ifcmderror_typ

TREE	Branch list	edge_zweigliste

Comment:

The Branch list TREE must contain a sequence of one or two Pet_Else constructs. The first Pet_Else construct will be used as a Then branch.

61. PET_IFBLOCKERROR

elementTyp = node_pet_ifblockerror_typ

string1 = label

TREE	Branch list	edge_zweigliste

Comment:

The Branch list TREE must contain a sequence of one or two Pet_Else constructs. The first Pet_Else construct will be used as a Then branch.

62. XOR

elementTyp = node_xor_typ

TREE	Operands	edge_bedingung

Comment: See AND

63. PET_PROC

elementTyp = node_pet_proc_typ

TREE	Option	edge_petoption
TREE	Param	edge_petparam
TREE	Internal Proc.	edge_intproc
TREE	Body	edge_rumpf
TREE	Abnormal	edge_abnorm

64. PET_IFBRANCH

elementTyp = node_pet_ifbranch_typ

TREE	Condition	edge_bedingung
TREE	Alternative	edge_alternative

65. CLASS

elementTyp = node_class_typ

TREE	Header	edge_text
TREE	Body	edge_rumpf
TREE	Footer	edge_fuss

Comment:

The Footer TREE as well as the Header TREE may contain exactly one comment or dummy construct..

66. PRIVATE

elementTyp = node_private_typ

Comment:

This construct does not require any parameters or subTREEs. It may occur only in the body of a class construct.

67. PUBLIC

elementTyp = node_public_typ

Comment:

This construct does not require any parameters or subTREEs. It may occur only in the body of a class construct.

68. PROTECTED

elementTyp = node_protected_typ

Comment:

This construct does not require any parameters or subTREEs. It may occur only in the body of a class construct.

69. PROG_AUFRUF

elementTyp = node_prog_aufruf_typ

string1 = info_string

Parameters:

info_string ... All information concerning this construct will be packed into an info string by the parser. This string will be analyzed in detail by EasyCODE.

For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

70. BTMITTEL

elementTyp = node_btmittel_typ

string1 = info_string

Parameters:

info_string ... All information concerning this construct will be packed into an info string by the parser. This string will be analyzed in detail by EasyCODE.

For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

71. VARIANTE

elementTyp = node_variante_typ

string1 = info_string

Parameters:

info_string ... All information concerning this construct will be packed into an info string by the parser. This string will be analyzed in detail by EasyCODE.

For a description of the info strings see Appendix A of the documentation concerning the ETF file format (ETF.RTF file in the installation directory).

72. ASS_THRU

elementTyp = node_ass_thru_typ

string1 = label

TREE	Expression	edge_text
TREE	Body	edge_rumpf

73. ASS_CYCLE

elementTyp = node_ass_cycle_typ

string1 = label

TREE	Reg	edge_text
TREE	Body	edge_rumpf

74. ASS_CASE

elementTyp = node_ass_case_typ

string1 = label

TREE	Reg	edge_text
TREE	Branch list	edge_zweigliste

75. ASS_EXIT

elementTyp = node_ass_exit_typ

string1 = label

TREE	Param	edge_text

76. PET_INTPROC

elementTyp = node_pet_intproc_typ

TREE	Header	edge_text
TREE	Option	edge_petoption
TREE	Param	edge_petparam
TREE	Body	edge_rumpf
TREE	Abnormal	edge_abnorm

77. FUNC

elementTyp = node_func_typ

TREE	Header	edge_text
TREE	Body	edge_rumpf

19Use of Constructs

Each of the various EasyCODE components provides only some of the constructs. The write_element calls must be restricted to the available constructs, since otherwise unpredictable errors may occur.[9]

In the following tables, the menu items available in the Insert menus of the individual EasyCODE components are assigned to the corresponding constructs.

SPX

Construct in the Insert menu	Internal construct name
Statement	Comment
IF-THEN-ELSE	If
SWITCH	Case
WHEN	Casebranch
CASE	Switch
OF	Switchbranch
FOR	For
WHILE	While
REPEAT	Repeat
LOOP	Cycle
EXIT	Break
Procedure	Proc
Procedure call	Call
Function	Func
Block	Block
Frame	Rahmen
AND	And
OR	Or
NOT	Not
Condition	Comment[10]

DS

Construct in the Insert menu	Internal construct name
Object	Proc
Data element	Comment
Iteration	Wieder
Option	Wieder
Selection	Auswahl
Alternative	Default

[9] In some cases, EasyCODE components may also display constructs not available in the component concerned, but not all constructs were implemented in all components.

[10] When a condition is inserted, a dummy will be inserted, which will become a comment when edited.

COB

Construct in the Insert menu	Internal construct name
Statement	Comment
Exception	COB_Exception
Cobol program	COB_Programm
SECTION	COB_Section
PARAGRAPH	COB_Paragraph
PERFORM > Inline	COB_Inline
PERFORM > Outline	Call
PERFORM > TIMES	COB_Times
PERFORM > TEST BEFORE	While
PERFORM > TEST AFTER	Repeat
PERFORM > BEFORE VARYING	For
PERFORM > AFTER VARYING	COB_Varyingafter
IF-THEN-ELSE	If
EVALUATE	COB_Evaluate
WHEN expression	Switchbranch
SEARCH	COB_Search
WHEN condition	Casebranch
CALL	COB_Call
ENTRY	Entry
GOBACK	Detach
EXIT	Exit
EXIT > PERFORM	COB_Exitper
EXIT > TO TEST	COB_Exittest
EXIT > PROGRAM	COB_Exitprog

CPP

Construct in the Insert menu	Internal construct name
Statement	Comment
if	If
switch	C_Switch
case	C_Case
default	Default
for	For
while	While
do while	Repeat
Class	Class
private	Private
public	Public
protected	Protected
Function	Proc
Block	Block
break	Exit
continue	Detach
return	Return

20 File-Specific Information

This section deals with information concerning the graphic representation of the structure diagram, such as fonts or the IF layout. This information is saved together with the EasyCODE file and will be restored when the file is opened again.

This information can also be saved in the source and restored during file analysis. Like a construct, the information is returned by the write_element function, with special types being used not corresponding to a construct. The EasyCODE settings are only modified after a write_element call with the corresponding information and a successful file analysis. The write_element calls with these types may occur at any time.

21 Short Info

elementTyp = node_kurzinfo

string1 = text

Parameters:

text	...	Text for the short info. The same restrictions apply as those applying to all other types of text delivered via parser interface.

Comment:

The short info is a short text describing the file contents. It can be entered into the Save as dialog window and will be displayed in the Open dialog window when EasyCODE files are selected.

22 IF Layout

elementTyp = node_if

par1 = vertical (BOOL)

Parameters:

vertical	...	TRUE, if the vertical layout of an IF is required.

23 Segment Numbers

elementTyp = node_levelnumbers

par1 = on (BOOL)

Parameters:

on	...	TRUE, if segment numbers are to be displayed

24 Line Numbers

elementTyp = node_linenumbers

par1 = on (BOOL)

Parameters:

on	...	TRUE, if line numbers are to be displayed

25 Screen Font

elementTyp = node_screenfont

string1 = info_string

Parameters:

info_string	...	One-line text describing the screen font. The format of the text is the same as that of the line in the INI file, when the font settings are saved.

Comment:

The format of the info_string consists of a sequence of LOGFONT and CHOOSEFONT structure elements in the form of text, which are separated by a comma. For the definitions of the structures, see the SDK documentation. The following list is displayed in several lines, so that it is easier to survey. The info_string must, however, not contain any line breaks.

```
LOGFONT.lfFaceName,
CHOOSEFONT.lpszStyle,
CHOOSEFONT.iPointSize,
CHOOSEFONT.nFontType,
LOGFONT.lfHeight,
LOGFONT.lfWidth,
LOGFONT.lfWeight,
LOGFONT.lfItalic,
LOGFONT.lfEscapement,
LOGFONT.lfOrientation,
LOGFONT.lfUnderline,
LOGFONT.lfStrikeOut,
LOGFONT.lfCharSet,
LOGFONT.lfOutPrecision,
LOGFONT.lfClipPrecision,
LOGFONT.lfQuality,
LOGFONT.lfPitchAndFamily
```

26 Printer Font

elementTyp = node_printerfont

string1 = info_string

Parameters:

info_string	...	One-line text describing the printer font. The format of the text is the same as that of the line in the INI file, when the font settings are saved.

Comment:

See Screen Font.

27 Highest Level-ID

elementTyp = node_lastlevelid

string1 = levelid

Parameters:

levelid	...	Highest level-ID occurring in the source in text form.

Comment:

Every segment has an individual ID required for OLE. This ID is specified with write_level_start. To be able to assign other IDs, EasyCODE must know the last ID.

If level-IDs are specified with write_level_start, the highest level-ID must be delivered with node_lastlevelid anytime during the parser run. Usually, the highest level-ID is stored in the source.

If the level-IDs are set to 0 at write_level_start, the highest level-ID must not be delivered with node_lastlevelid during the parser run.

Since the level-ID is of the DWORD type, it will be converted to text form and delivered as string1.

28 Examples

The following examples illustrate the correct sequence of function calls. The examples 1 and 2 were created with a Clipper parser, example 3 was created with a BNF parser. For every source code, the corresponding functions and the corresponding structure diagram are shown.

EasyCODE counts every write_tree_start and write_tree_end in the open TREEs. The counting begins with 0 and is increased by 1 each time a write_tree_start is encountered and reduced by 1 each time a write_tree_end is encountered. The result will be displayed below the line containing the corresponding function call, so that it is easier to find corresponding function calls.

Analogously, every write_level_start and write_level_end is counted and the result displayed.

Example 1.

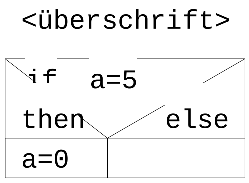
Source

```
IF a=5
  a=0
ENDIF
```

Function calls

```
+++ write_tree_start  (predTyp=node_level_typ edge=edge_rumpf)
0
>>> write_level_start (line=0, levelID=0,
0                     (posOfLevel receives 38)
+++ write_tree_start  (node_level_typ, edge_rumpf)
1
... write_element     (type=node_if_typ,line=1
                       par1=0, par2=0)
+++ write_tree_start  (predTyp=node_if_typ edge=edge_bedingung)
2
... write_element     (type=node_comment_typ,line=1
                       par1=0, par2=0
                       string1="a=5")
+++ write_tree_end()
2
+++ write_tree_start  (predTyp=node_if_typ edge=edge_then)
2
... write_element     (type=node_comment_typ, line=2
                       par1=0, par2=0
                       string1="a=0")
+++ write_tree_end()
2
+++ write_tree_start  (predTyp=node_if_typ edge=edge_else)
2
... write_element     (type=node_dummy_typ, line=3
                       par1=node_if_typ, par2=edge_else)
+++ write_tree_end()
2
+++ write_tree_end()
1
>>> write_level_end   (posOfLevel=38, posSource=0
0                      lengthSource=0, lines=3)
+++ write_tree_end()
0
```

Structure Diagram



Example 2.

Source

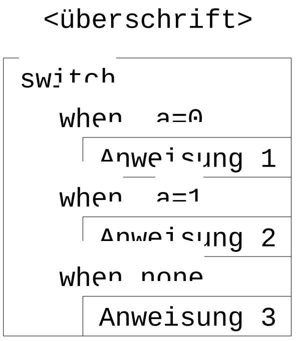
```
DO CASE
   CASE a=0
      Statement 1
   CASE a=1
      Statement 2
   OTHERWISE
      Statement 3
ENDCASE
```

Function calls

```
+++ write_tree_start  (predTyp=node_level_typ edge=edge_rumpf)
0
>>> write_level_start (line=0, levelID=0,
0                      posOfLevel receives 38)
+++ write_tree_start  (predTyp=node_level_typ edge=edge_rumpf)
1
... write_element     (type=node_case_typ, line=2
                       par1=0, par2=0)
+++ write_tree_start  (predTyp=node_case_typ edge=edge_zweigliste)
2
... write_element     (type=node_casebranch_typ, line=2
                       par1=0, par2=0)
+++ write_tree_start  (predTyp=node_casebranch_typ
                       edge=edge_bedingung)
3
... write_element     (type=node_comment_typ, line=2
                       par1=0, par2=0
                       string1="a=0")
+++ write_tree_end()
3
+++ write_tree_start  (predTyp=node_casebranch_typ
                       edge=edge_alternative)
3
... write_element     (type=node_comment_typ, line=3
                       par1=0, par2=0
                       string1="Statement 1")
+++ write_tree_end()
3
... write_element     (type=node_casebranch_typ, line=4
                       par1=0, par2=0)
+++ write_tree_start  (predTyp=node_casebranch_typ
                       edge=edge_bedingung)
3
... write_element     (type=node_comment_typ, line=4
                       par1=0, par2=0
                       string1="a=1")
+++ write_tree_end()
3
```

```
+++ write_tree_start  (predTyp=node_casebranch_typ
                       edge=edge_alternative)
3
... write_element     (type=node_comment_typ, line=6
                       par1=0, par2=0
                       string1="Statement 2")
+++ write_tree_end()
3
+++ write_tree_end()
2
+++ write_tree_start  (predTyp=node_case_typ edge=edge_ofrest)
2
... write_element     (type=node_casebranch_typ, line=6
                       par1=0, par2=0)
+++ write_tree_start  (predTyp=node_casebranch_typ
                       edge=edge_bedingung)
3
... write_element     (type=node_dummy_typ, line=6
                       par1=node_casebranch_typ, par2=edge_bedingung)
+++ write_tree_end()
3
+++ write_tree_start  (predTyp=node_casebranch_typ
                       edge=edge_alternative)
3
... write_element     (type=node_comment_typ, line=7
                       par1=0, par2=0
                       string1="Statement 3")
+++ write_tree_end()
3
+++ write_tree_end()
2
+++ write_tree_end()
1
>>> write_level_end   (posOfLevel=38, posSource=0
0                      lengthSource=0, lines=8)
+++ write_tree_end()
0
```

Structure Diagram



Example 3

Source

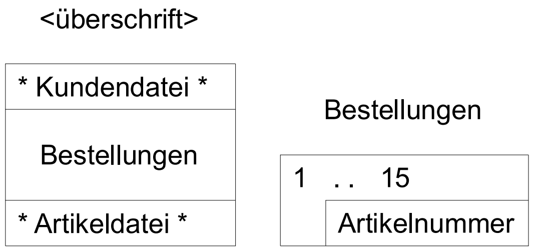
```
* EasyCODE(DS) ( 1
*
* Customer file *
* EasyCODE(DS) ( 2
Orders *
1{Article number}15
* EasyCODE(DS) ) *
* Article file *
* EasyCODE(DS) ) *
```

This is an example from EasyCODE(DS). The "*EasyCODE(DS) ("comment line marks a new segment. The top level segment is also indicated by this line.

Function calls

```
+++ write_tree_start  (predTyp=node_level_typ edge=edge_rumpf)
0
>>> write_level_start (line=2, levelID=1,
0                      posOfLevel receives 38)
+++ write_tree_start  (predTyp=node_level_typ edge=edge_rumpf)
1
... write_element     (type=node_comment_typ, line=3
                       par1=0, par2=0
                       string1="* Customer file*")
>>> write_level_start (line=5, levelID=2,
1                      posOfLevel receives 80
                       ebenenKommentar.line=5
                       text="Orders")
+++ write_tree_start  (predTyp=node_level_typ edge=edge_rumpf)
2
... write_element     (type=node_wieder_typ, line=6
                       par1=0, par2=0)
+++ write_tree_start  (predTyp=node_wieder_typ edge=edge_text)
3
... write_element     (type=node_comment_typ, line=6
                       par1=0, par2=0
                       string1="1")
+++ write_tree_end()
3
+++ write_tree_start  (predTyp=node_wieder_typ edge=edge_rumpf)
3
... write_element     (type=node_comment_typ, line=6
                       par1=0, par2=0
                       string1="Article number")
+++ write_tree_end()
3
+++ write_tree_start  (predTyp=node_wieder_typ edge=edge_max)
3
... write_element     (type=node_comment_typ, line=6
                       par1=0, par2=0
                       string1="15")
+++ write_tree_end()
3
+++ write_tree_end()
2
>>> write_level_end   (posOfLevel=80, posSource=0
1                      lengthSource=0, lines=0)
... write_element     (type=node_comment_typ, line=8
                       par1=0, par2=0
                       string1="* Article file*")
+++ write_tree_end()
1
>>> write_level_end   (posOfLevel=38, posSource=0
0                      lengthSource=0, lines=0)
+++ write_tree_end()
0
```

Structure Diagram



Index

A

Action block....23
AGBLOCK....23
AND....20
ANSI....5; 9; 10
ASS_CASE....30
ASS_CYCLE....30
ASS_EXIT....30
ASS_THRU....29
AUSWAHL....26

B

BLOCK....21
BREAK....19
BS2....18
BTMITTEL....29

C

C_CASE....24
C_SWITCH....23
CALL....22
CASE....20
CASEBRANCH....20
Character set....5; 9; 10
CLASS....28
COB - use of constructs....32
COB_CALL....25
COB_EVALUATE....26
COB_EXCEPTION....25
COB_EXITPER....25
COB_EXITPROG....25
COB_EXITTEST....25
COB_INLINE....24
COB_PARAGRAPH....24
COB_PROGRAMM....24
COB_SEARCH....26
COB_SECTION....24
COB_TIMES....25
COB_VARYINGAFTER....25
COMMENT....21
Comment construct....5
COND....20
Configuration file....4
Constructs....18
CPP - use of constructs....32
CYCLE....19

D

Debugging....16
DEFAULT....24
DETACH....23
DOS source files....5
DS - use of constructs....31
DUMMY....18

E

easy_parse....7
ENTRY....26
error file....15
Error messages....15
Examples....35
EXIT....22
Exit perform....25
Exit program....25

F

File-specific information....33
FOR....22
FREE FORMAT....23
FUNC....30
Functionality....5
Functions....7

H

Highest level-ID....34

I

IF 19
IF layout....33
IFERROR....23
INI file....4
Inline Perform....24
ISP....23

J

JETPROC....23

L

LEAVE....23
level....10; 21
Line numbers....33

N

NOT....20

O

OEM....9
OR....20

P

parse_error..13
Parser configuration...17
Parser DLL...4
Perform Times..25
Perform varying after...25
PET_AGBLOCK..27
PET_BLOCK..27
PET_ELSE...27
PET_FOR...27
PET_IF..27
PET_IFBLOCKERROR...28
PET_IFBRANCH..28
PET_IFCMDERROR...28
PET_INTPROC...30
PET_JUMPRESTART..27
PET_PROC..28
PET_REPEAT..27
PET_WHILE..27
Printer font..34
PRIVATE...28
PROC..26
PROG_AUFRUF..29
PROTECTED...29
PUBLIC..29

R

RAHMEN...26
REPEAT..22
RETURN..24

S

Screen font..33
Segment...5
Segment numbers..33
Short info..33
Source code
  analysis...4
SP, SPX - use of constructs....................................31
SWITCH...22
SWITCHBRANCH..22

T

TREE...5

U

Unix source files..5
Use of constructs...31

V

VARIABLE..24
VARIANTE..29

W

warnings..15
WHEN...22
WHILE..19
WIEDER...26
write_debug..14
write_element..9
write_level_end...12
write_level_start..10
write_tree_end...13
write_tree_start..12

X

XOR...28