CLiP - *C* ode from *Li* terate *P* rograms

Provisional User Manual

Disclaimer:

This document is meant for programmers who are already familiar with the literate programming paradigm. It pretends in no way to be a complete user manual in the real meaning of the word. But the document should supply sufficient hints to experiment successfully with the CLiP system.

You are assumed to be familiar with the ideas and design goals of the CLiP system as explained in [Ammers and Kramer 1993]. Also you should have read `READ_ME.TXT` or `READ_ME.PS` which come with the executables and understand the meaning of the INI-file which guides the extraction process.

The DOS and VMS versions of CLiP consist of two independent programs, CLIP_1 and CLIP_2. CLIP_1 is purely to create an INI-file for CLIP_2, which performs the actual work. The Unix version consists of only one program, CLIP (which is identical to CLIP_2), and an INI-file should be made using the supplied scripts or an editor. This manual only deals with the second program.

1 Introduction

In short the idea behind CLiP is to define a style of programming sufficiently formal to be recognized by a literate programming automaton. This syntax (which we will refer to as the CLiP-syntax) should not obstruct a natural way of programming. Ideally the system should be smart enough to "see" stubs, slots and the like by "inspecting" the documentation and simply "understanding" the refinements the programmer has made. Alas, this is far beyond the current potential of artificial intelligence and we will have to reach the automaton a helping hand by defining a special syntax.

So we want the "look and feel" of a documentation to be as "natural" as possible, but with CLiP features that can be recognized by an automaton. On the other hand should the reader be burdened as little as possible with the fact that additional processing is needed to extract the modules. In systems like VAMP [Ammers 1984] and WEB [Knuth 1984] - which use batch formatters - this is achieved by adding explicit commands in the source-files that are suppressed in the formatted documentation. But this technique does not work if the documentation environment is an interactive word-processor (Word Perfect, Lotus Manuscript, MS-Word, Ami Pro, etc.). So CLiP uses either the ASCII-export from a word-processors or the ASCII input of a formatter and extracts modules from there. In these files CLiP recognizes the important sections by a particular programming-style rather than by explicit commands.

The first section describes the syntax that CLiP "sees". The next section gives a couple of useful examples. In section 4 we describe the options that may be used to customize the process of module generation. In section 5 we give some hints on how to organize your working with CLiP efficiently. Stubs and slots my remaining sections illustrate the style of documentation and programs that would result. CLiP is in development and we compile the known bugs in section 6.

2 Syntax (programming style)

CLiP-lines – that is lines which have a special meaning to CLiP – start with a CLIP-left-parenthesis (CLIP-lpar) and end with a CLIP-right-parenthesis (CLIP-rpar). Both tokens are constructed as extensions of the tokens encapsulating comments which we will refer to as `COMM-START` and `COMM-END`. If we program in Pascal then everything between "`(*`" and "`*)`" is seen as comment, thus we have the convention:

```
COMM-START = "(*"
COMM-END   = "*)"
```

Extension of the comment token is by a character with a special meaning to clip, the `CLIP-CHAR`. In our case we assume

```
CLIP-CHAR = "*"
```

Thus CLiP-lines are enclosed by

CLIP-lpar of the form "`(**`"

and

CLIP-rpar of the form "`**)`"

The parameters `COMM-START`, `COMM-END` and `CLIP-CHAR` are set by the INI-file and it is evident that the convention can be adjusted.

CLiP distinguishes six source line categories:

```
1 | (*******************  <arbitrary string>  *********************)
2 | (***************  End of <arbitrary string>  *****************)
3 | (**                       <arbitrary string>                 **)
4 | (**************************************************************)
5 | <the empty string>
6 | <any string not falling in one of the above categories>
```

We will refer to a line from the second category as an "L2" and so on and investigate the meaning is of the various categories for CLiP.

The string "`End of`" of an L2 has a special meaning that distinguishes an L2 from an L1. Again this special string is defined by a parameter in the INI-file

```
END-STRING = "ENDOF"
```

Observe that CLiP does not distinguish between upper and lower case and forgets about spaces. In fact CLiP reduces the <arbitrary string> internally to a sequence of the characters `"A"`-`"Z"`, `"0"`-`"9"` and `"."`

Note that for an L3 the character immediately following the `CLIP-CHAR` on the left side and the character preceding the `CLIP-CHAR` on the right side may be anything apart from the `CLIP-CHAR` itself.

2.1 Stub blocks and documentation blocks

From CLiP's point of view the source-files are divided in *documentation blocks* and *stub-blocks*. A stub-block starts with an L1 and usually ends with an L2. A special kind of stub, a so called quick stub (see sect. 3.3 and 4), is ended by an L5 or an <EOF>. Everything outside a stub-block is a documentation block and ignored by CLiP. For this reason we say that CLiP is in *active mode* during the processing of a stub-block and in *passive mode* otherwise. The only way to change CLiP from passive to active mode is by an L1 and the usual way from active to passive is through an L2. An L2 in passive mode is presumably an error and will be ignored.

2.2 Segments

A stub-block contains one and only one *stub-segment*, which starts the stub-block. The stub-segment may be followed by any number of *slot-segments* and/or *code-segments*.

An L1 always starts a new stub- or slot-segment and the segment continues with any number of lines of type L3 or L4. A segment is a slot-segment if and only if it is not the first one of a stub-block. A code-segment consists of any number of contiguous lines of type L5. Stub- and slot-segments have an *identification* or *name* which is constructed somehow from the `<arbitrary string>`s inside the segment.

L3 and L4 are continuations of stub- and slot-segments, but with a slightly different status. An L3 that cannot be pasted to a segment is flagged as an error. But an L4 under the same conditions is interpreted as belonging to the current code-segment.

2.3 Options

The module extraction process can be tuned by means of *options*. Stub-segments as well as slot-segments may have options. An option starts with a keyword signalled by a special `OPTION-MARKER`. The option keyword may be abbreviated to a unique headerstring for that option. We will assume `"#"` for `OPTION-MARKER`, but of course this again is a parameter set by the INI-file. The argument(s) following an option continue till the next option-keyword or till the end of the segment. Therefore the name of a segment must precede the options.

In general options operate on the *inside* of the stub only, since that is the part of the program that the programmer of this particular refinement has in his or her mind. In addition options can be inherited. The meaning of the various options is explained in section 4. Some frequently used options show up in the examples of section 3.

3 Examples

In the examples we will illustrate the most important features of CLiP by making remarks on excerpts drawn from the documentation of CLiPs predecessor, VAMP.

3.1 Example 1: Defining a module

The module VAMP:
Starting from nowhere, the empty program will do.

```
 1 (**** #File "VAMP.PAS". #Indent on.                          ****)
 2
 3 (****************************************************************)
 4 (* Routine:      VAMP    -   Main module of the VAMP system.    *)
 5 (* Purpose:      Main module and unique entry point to the      *)
 6 (*               VAMP-system.                                   *)
 7 (* Interface:    TTY -   All communication with the user        *)
 8 (*               proceeds via the terminal.                     *)
 9 (* Author/Date: VAMP project management, Sept. 12, 1983.        *)
10 (****************************************************************)
11
12 (**********************  VAMP (body)  *************************)
13
14 (***********************  End of VAMP  ************************)
15
16
18 (*********  VAMP (body) #def  ***********)
19 PROGRAM     VAMP (INPUT, OUTPUT);
20     BEGIN
21     WRITELN ('!!! VAMP was here !!!');
22     END (*VAMP*).
23 (**********  End of VAMP (body)  *********)
```

Remarks:

There are two stubs in this section. Line 1-14 and line 18-23. Both stub-blocks have a stub-segment of one line only (lines 1 and 18 respectively).

The first stub-segment has no identification. It specifies the start of a new output module by the FILE option and thus is not meant to be referenced (see also example 5).

Line 2 (an L5) separates line 3 (an L4) from the stub-segment and makes it belong to the code-segment which expands over lines 2-11.

There resides only one slot inside the first stub (line 12) and this slot is identified as "`VAMPBODY`". The second stub has no slots at all.

Line 13 is a trivial code-segment and line 14 marks the end of the first stub-block. The string following the "End of" is optional and does not have to match the stub name in any way.

3.2 Example 2: Multiple slots

Environment module:

In the past there have been made several changes in the VAMP program. These changes have been made in the extracted modules rather then in the VAMP source. To make source files consistent with the modules, the source files have been updated in march 1990 by Jeroen Reef.

Furthermore, the updated source files contain two additional modules, VAMP.MSG and DECLAR_MOD. The module VAMP.MSG contains the error messages of VAMP and the module DECLAR_MOD contains the types and constants used by several modules of VAMP. This module uses the ENVIRONMENT, which directs the compiler to generate an environment file DECLAR.PEN. Other modules can reference the identifiers declared in DECLAR_MOD by inheriting the environment with the INHERIT attribute.

```
 1 (******      #File "DECLAR_MOD.PAS". #Indent ON.          ******)
 2
 3 (**************************************************************)
 4 (* Module to contain all parameters and global declarations    *)
 5 (* of the VAMP system.                                         *)
 6 (**************************************************************)
 7 [ENVIRONMENT ('DECLAR.PEN')]    MODULE DECLARS;
 8
 9 (**********  Parameters of the VAMP-system (#mul)  *************)
10
11 TYPE
12     ABSTRACT  = (DEFINED, UNDEFINED);
13     (*********  Simple types of the VAMP-system (#mul)  ********)
14     (********  Structured types of the VAMP-system (#mul) ******)
15 END.
16 (*****************  End of DECLARE_MOD.PAS  *******************)
```

Remarks:

The stub-block extends over 16 lines. Line 1 is the stub-segment and line 16 closes the stub-block. Lines 2-8 constitute the first code-segment of the stub, lines 10-12 the middle one and line 15 the last one.

There are three slots (or slot-segments), lines 9, 13 and 14. All slots consist of one line only and carry the multiple option. For instance the first slot can swallow any number of parameter definition that might turn up in the future.

The names (identifications) of the slots are respectively

```
"PARAMETERSOFTHEVAMPSYSTEM"
"SIMPLETYPESOFTHEVAMPSYSTEM"
"STRUCTUREDTYPESOFTHEVAMPSYSTEM"
```

3.3 Example 3: Quick stubs and abstract data types

First level data structures:

This level introduces the data-structure TTY_INFO, containing the initial communication between the terminal and the program. The definition requires in addition four system parameters and some constants and types which are related to file specifications. The structure of CODE_INFO is left open for the time being.

```
 1 (**********  Parameters of the VAMP-system (#quick)  ***********)
 2 (*          --------  Parameters of TTY_INFO  ---------        *)
 3 MAX_FILE_SPEC_L =  255;      (* Maximum length file-specific.   *)
 4 MAX_FILE_NAME_L =   39;      (* Maximum length file-name.       *)
 5 MAX_FILE_EXT_L  =   39;      (* Maximum length file-extension.  *)
 6 MAX_IN_FILES    =    8;      (* Maximum number of in-files for  *)
 7                              (* a single run.                   *)
 8 UPB_IN_FILES    =   9;       (* = MAX_IN_FILES + 1.             *)
 9 MAX_MODULES     =   10;      (* Maximum number of modules       *)
10                              (* specified for a run.            *)
11 UPB_MODULES     =   11;      (* = MAX_MODULES + 1.              *)
12 EMPTY           = '';
13
14 (*********  Simple types of the VAMP-system (#quick)  **********)
15 (*          --------  Simple types of TTY_INFO  ---------      *)
16 FILE_SPEC = VARYING [MAX_FILE_SPEC_L] OF CHAR;
17 FILE_NAME = VARYING [MAX_FILE_NAME_L] OF CHAR;
18 FILE_EXT  = VARYING [MAX_FILE_EXT_L]  OF CHAR;
19
20 (********  Structured types of the VAMP-system  ****************)
21 (*      --------  Structured types of TTY_INFO  ---------      *)
22 TTY_INFO =  RECORD
23                 IN_FILES:   ARRAY [1..UPB_IN_FILES] OF FILE_SPEC;
24                 MODULES:    ARRAY [1..UPB_MODULES] OF FILE_NAME;
25                 DFLT_EXT:   FILE_EXT;
26                 INV_MODE,
27                 GO:         BOOLEAN;
29             END (*RECORD*);
30 (****************  Declaration of CODE_INFO  ******************)
31 (********  End of Structured types of the VAMP-system  *********)
32
33
34 (*************  Declaration of CODE_INFO (#def)  ***************)
35 CODE_INFO = ABSTRACT;
36 (********************  End of declaration  ********************)
```

Remarks:

This section contains five stubs altogether, i.e. lines 1-12, 14-18, 20-31 and 34-36. The first two blocks one have no internal slots. The quick option promotes them to quick stubs, which means that they are completed by the first line that is not an L6.

The third stub (20-31) illustrates how an abstract data type can be implemented. The stub is an ordinary one with one slot - identified as "`DECLARATIONOFCODEINFO`" - at line 30. The slot is by default satisfied with the stub at lines 34-36. But this declaration will be replaced by a new one at some time in the future.

3.4 Example 4: An ordinary refinement step

Body of VAMP:

The input-files will be processed sequentially in the same order as given by the user. The intermediate file "VAMP.TMP" is guarded by a sentinel to simplify backspacing later on and it needs an additional global declaration.

```
    (***********   Simple types of the VAMP-system  ************)
    FTYPE      = FILE OF CHAR;
    (******************  End of declaration   *****************)
```

Now the body of VAMP expands to

```
(**********************  VAMP (body)  *************************)
[INHERIT (’SYS$LIBRARY:STARLET.PEN’, ’DECLAR.PEN’)]
PROGRAM     VAMP (INPUT, OUTPUT);

(*****************  Constants of VAMP (#mult)  ****************)
(*****************  Types of VAMP (#mult)  ********************)
VAR
    FILE_CNT:       1..UPB_IN_FILES;
    CURR_IN_FILE:   TEXT;    (* Currently read file.           *)
    CODE_LINES:     FTYPE;   (* From input extracted code.     *)
    TTY_DATA:       TTY_INFO;
    CODE_STRUCT:    CODE_INFO;
    (************  Variables of VAMP (#mult)  *****************)

[EXTERNAL] PROCEDURE    ASKTTY (VAR TTY_DATA: TTY_INFO);  EXTERN;
(***************  Functions of VAMP (#mult)  ******************)

BEGIN
    (* Take the data, that are needed for this VAMP run from   *)
    (* the terminal.                                           *)
    ASKTTY (TTY_DATA);
    WITH TTY_DATA DO
    IF GO THEN
    BEGIN
        (********************  VAMP (C)  *********************)
        (** Initialize CODE_STRUCT.                          **)
        (*****************************************************)

        OPEN (CODE_LINES, FILE_NAME :=’VAMP.TMP’,
              ORGANIZATION := RELATIVE, ACCESS_METHOD := DIRECT,
              DISPOSITION := DELETE);
        REWRITE (CODE_LINES);
        FILE_CNT := 1;
        WHILE (IN_FILES [FILE_CNT] <> EMPTY) DO
        BEGIN
            WRITELN (’Proceeding on file ’,IN_FILES [FILE_CNT]);
            OPEN (CURR_IN_FILE, FILE_NAME := IN_FILES [FILE_CNT],
                  HISTORY := READONLY);
            RESET (CURR_IN_FILE);
            PUT (CODE_LINES);   (* Start file with a sentinel. *)

```

```
45             (*****************  VAMP (A)  ********************)
46             (** Build CODE_STRUCT and fill CODE_LINES by a    **)
47             (** scan of CURR_IN_FILE, using the information   **)
48             (** of MODULES, DFLT_EXT and INV_MODE.            **)
49             (************************************************)
50
51             CLOSE (CURR_IN_FILE);
52             FILE_CNT := FILE_CNT + 1;
53         END (*WHILE*);
54
55         (*******************  VAMP (B)  *********************)
56         (** Generate the files as specified by CODE_STRUCT    **)
57         (** from the data contained by CODE_LINES.            **)
58         (******************************************************)
59
60         (* Close and Delete scratch-file CODE_LINES.           *)
61         CLOSE (CODE_LINES, DELETE);
62     END
63     ELSE
64         WRITELN ('!!! You specified an empty run - try again !!!');
65 END (*VAMP*).
66 (******************  End of VAMP (body)  *********************)
67
68
69 (************  Constants of VAMP (#leader, #quick)  ************)
70 CONST
71 (************  Types of VAMP (#leader, #quick)  ***************)
72 TYPE
```

Remarks:

This refinement contains four stubs (lines 1-3, 4-66, 69-70 and 71-72). The second stub has seven slots (lines 8, 9, 16, 19, 28-30, 45-49 and 55-58). Lines 22, 23 and 60 are simple code-lines from CLiPs point of view.

The leader option of the last two stubs defines the code that will be inserted in front of a stub that matches the slot.

3.5 Example 5: A subroutine module

```
(******        #File "ASKTTY.PAS"                         ******)
[INHERIT ('DECLAR.PEN')]        MODULE ASKTTY (INPUT, OUTPUT);

(***********  External procedures of ASKTTY (#mult) ************)

(***************************************************************)
(* Routine:     ASKTTY  -   ASK information from TTY.          *)
(* Purpose:     To obtain from the TTY the information which   *)
(*              is needed to perform a VAMP run.               *)
(* Interface:   TTY_DATA -  Data from TTY to VAMP.             *)
(*              TTY -       Source of all knowledge.           *)
(* Author/Date: VAMP project management, September 29, 1983.   *)
(***************************************************************)
[GLOBAL] PROCEDURE  ASKTTY (VAR TTY_DATA: TTY_INFO);

(*************  Constants of ASKTTY (#multiple)  ***************)
(*************  Types of ASKTTY (#multiple)  *******************)
(*************  Variables of ASKTTY (#multiple)  ***************)
(*************  Procedures of ASKTTY (#multiple)  **************)

BEGIN
    WITH TTY_DATA DO
    BEGIN
        (********************  ASKTTY (1)  *********************)
        (** Get IN_FILES from TTY.                            **)
        (*******************************************************)

        GO := NOT (IN_FILES[1] = EMPTY);
        IF GO THEN
        BEGIN
            (****************  ASKTTY (2)  *********************)
            (** Ask which modules must be generated by VAMP.  **)
            (** Set INV_MODE, MODULES and GO accordingly.     **)
            (***************************************************)
        END (*IF*);
        IF GO THEN
        BEGIN
            (****************  ASKTTY (3)  *********************)
            (** Ask DFLT_EXT from the terminal.               **)
            (***************************************************)
        END (*IF*);
    END (*WITH*);
END (*ASKTTY*);

END (*MODULE*).
(**********************  End of MODULE  ************************)


(**********  Constants of ASKTTY (#leader, #quick)  ************)
CONST
```

```
51 (**********  Types of ASKTTY (#leader, #quick)  ****************)
52 TYPE
53 (**********  Variables of ASKTTY (#leader, #quick)  ************)
54 VAR
```

Remarks:

A clear cut example of a subroutine definition in VAX/VMS Pascal that is to be compiled independently. The refinement contains stubs at lines 1-46, 49-50, 51-52 and 53-54. The last three stubs are all quick stubs.

The first stub is a main stub. It carries no identification and only specifies the name of the file that will finally hold the module. Had the file-option be omitted, this would have meant an error.

Lines 4, 16, 17, 18 and 19 are single-line slot for future declarations. The leader stubs at the end secure syntactic details. Lines 24-26, 31-34 and 38-40 are ordinary slots again.

4 Semantics of options

#Comment (stub, slot)
The comment option transforms the special CLIP-characters to a predefined format which suits a particular programming language. The option has an obligatory argument, e.g. **PASCAL**, **FORTRAN**, **C**, **ADA** etc. or **ON/OFF**. The ON/OFF argument specifies wether or not the slot- or stub-segment is to be included upon substitution. Comment options can be nested and the most local version controls the operation.

#Default (stub)
A default stub is prefixed to exclusively substituted if no other stub is found for that particular slot.

#Indent (stub, slot)
The indent option controls the indentation of the generated listing. Indent options can be nested and the most local option overrules the more global ones. Indent has as one optional argument, **ON** or **OFF**.

#File (stub)
The file option identifies the stub as a main stub (viz. the root of a new module). The option carries a string in quotes as argument. This string specifies the name of the file to be generated.

#Leader (stub)
The leader stub is usually combined with multiple slots. It modifies the environment of the stubs that will be substituted by inserting the leader stub in front of the first encountered normal stub.

#Multiple (slot)
A multiple slot accepts the substitution of any number of stubs.

#Optional (slot)
An optional slot accepts the substitution of 0 or 1 stubs only.

#Overrule (stub)
The overrule stub replaces the stub already substituted in a particular position. It is mainly meant for testing purposes.

#Quick (stub)
A quick stub is a stub without internal structure (viz. slots). Following the stub segment only L6-type lines are allowed. Any other type of line will end the stub.

#Separator (stub)
A separator stub is to be inserted between two consecutive stubs of the same slot.

#Trailer (stub)
The trailer stub is the counterpart of the default stub. It modifies the environment by adding the trailer stub after the last encountered normal stub.

5 Using CLiP

The complete CLiP system for DOS and VMS consists of two independent programs, CLIP_1.EXE and CLIP_2.EXE. The Unix version has only one program, CLIP, which is functionally identical to CLIP_2. CLIP_1 prepares a file, CLIP.INI, telling CLIP_2 (the actual module extractor) literally everything it has to know in order to perform a run (a detailed description of the structure of an INI-file is in the next section). Thus CLIP.INI specifies to CLIP_2 (CLIP):

- the files it has to read (i.e. the source files),
- the modules it should extract (i.e. target modules),
- the name of the report file,
- the syntax of the CLiP-lines (i.e. the style of the lines that trigger the module extraction process).

CLIP_1 allows a very detailed specification of the extraction process, much more detailed than you will need in general. For this reason the CLiP system comes with a couple of routines to shortcut CLIP_1 and for Unix users these routines are the primary tool to construct their INI-file. For a description of these routines we refer to section "Using CLiP" of the READ_ME file of this release.

When using CLiP it is recommended to introduce at least the following directories:

- A directory for the source files CLiP will use. This directory also holds the various INI-file you maintain to conduct the extraction process.
- A directory to contain the extracted modules.
- A directory to contain the results of the compilation and linking of the modules.
- In case CLiP is used in combination with a word-processor, it is wise to keep the word-processor files again in a separate directory. In this case the source file directory should be fed with ASCII exports from word-processor files.

6 The structure of an INI-file

This section explains the structure of an INI-file by means of an example file. You should understand this structure thoroughly before your try to edit INI-files yourself directly through an editor [Ammers 1993].

The example is an INI-file for MS-DOS. The only difference with other platforms is the specification of directories. The numbers at the beginning of every line are added for convenience of reference. They are not part of the file itself.

```
   <=================  Example of an INI-file  ====================>
 1 This file contains data that is needed to run CLiP
 2 And is generated by CLiP_MENU
 3 Modifying this file at own risk.
 4 Using CLiP_MENU is definitely recommended.
 5
 6 INTERACTIVE_MODE        Mode (INTERACTIVE/DEBUG/HELPFUL/AUTO)
 7 NO                      Interactive fault correction (YES/NO)
 8 BOTH                    Error message destination (TERMINAL/......)
 9 (*                      Left comment string
10 *)                      Right comment string
11 *                       Command character
12 ENDOF                   END string
13 #                       Option marker
14 EXTRACTED               The specified modules are (OMITTED/EX-
15 TRACTED)
16 ------------ REPORT FILE -----------
17 CLIP.RPT
18 ------------ INPUT FILES -----------
19 f:\LPT\MAN\EX01_A.ASC
20 f:\LPT\MAN\EX01_B.ASC
21 ------------ MODULES ---------------
22 F:\TEST\
23 PALINDRO.PAS
24 F:\TEST\
25 TESTDATA.IN
26 ------------ MODULE DIRECTORY ------
27 f:\LPT\
   ------------ END OF INI FILE -------
   <=================  End of Example INI-file  ===================>
```

Lines 1-5: Descriptive lines that are flushed upon reading.

Lines 6-14: Information behind position 24 is not interpreted.

Lines 6-7: These lines refer to unimplemented options. They should not be changed.

Lines 8: Identifies the output device for messages and reports. The first word of the line must be on of the following keywords TERMINAL, REPORTFILE (or FILE), BOTH or NONE and we assume the meaning self evident.

Lines 9-13: These lines together define the syntax of the lines CLiP will recognize. For an explanation of the CLiP syntax, refer to [Ammers 1993].

Lines 9: At most 6 characters to specify the "left comment string" i.e. the opening sequence of a comment string of the programming language you want to use. The characters cannot be letters (A-Z, a-z), digits (0-9) or a dot (.).

Lines 10: At most 6 characters to specify the "right comment string" i.e. the closing sequence of a comment string of the programming language you want to use. The characters cannot be letters (A-Z, a-z), digits (0-9) or a dot (.).

Lines 11: The character that is postfixed to a left comment string and prefixed to a right comment string in order to identify it as a special comment that CLiP has to process, a so called CLiP-line. The character cannot be a letter (A-Z, a-z), a digits (0-9) or a dot (.).

Lines 12: The leading characters that identify a particular CLiP-line as the end of a stub, i.e. a particular section that can be substituted elsewhere.

Lines 13: The character that identifies the options.

Line 14: Relates to the modules in the module section below. The specified modules must either be extracted or omitted and the first word of this line can only be OMITTED or EXTRACTED.

Line 15: Starts the section defining the file that mirrors the extraction process. This line should be copied literally.

Line 16: Path and file specification of the report file.

Line 17: Starts the section defining the input files for the extraction process, the so called source files. This line should be copied literally.

Line 18-19: Every line specifies a path and source file. There may be up to 64 lines in this section.

Line 20: Starts the section defining the modules that are considered during the extraction process. Whether the modules are extracted or omitted depends on line 14. This line should be copied literally.

Line 21-24: Every line pair specifies a path and a module file. There may be up to 64 pairs (128 lines) in this section.

Line 25: Starts the section to identify the default directory for modules, i.e. the directory where modules go to that have no explicit directory specified by the MODULES section. This line should be copied literally.

Line 26: Default directory for extracted modules.

Line 27: Identifies the end of the INI-file. This line should be copied literally.

7 Bugs, work-arounds and undocumented features

7.1 "SP_EXTR_STR: system failure .... call maintenance"

This failure may happen if the stubs have not been correctly closed with the END-STRING that has been specified as the syntax. The default value of this string is "ENDOF". Also the line containing the END-STRING should be properly closed. For example in a Pascal situation

```
(*********      End of module (2.1)        ***)
```

should be all right, but

```
(******      End of module (2.1)            **)
```

may give trouble since the string "**)" is incorrect here.

7.2 Options in general

Several options have been introduces with very advanced applications in mind. They have hardly been tested will probably not work. You won't need them either. For this reason don't use the options OVERRULE, SEPARATOR and TRAILER.

7.3 Multiple option

A slot with the "MULTIPLE" option raises an error if no stubs are found matching the slot. To prevent this error message one should specify the option "OPTIONAL" also for the slot.

7.4 Report file option NONE

This option to suppress the printing of reports to the terminal has not been implemented. It will be eliminated in some later version.

7.5 Generating target modules

7.5.1 Specification

If CLIP.INI specifies a target module that matches no file-option argument anywhere in the list of specified source files, CLiP does not complain. The matching is case sensitive and for instance a file-option argument

```
(**********  #file "EXAMPLE.PAS"  ***********************)
```

does not fit a module specification in CLIP.INI of the form

```
Example.PAS
```

This may lead to unpleasant surprises.

On MS-DOS and VAX/VMS platforms the specification of files is not context sensitive. For this reason the CLIP_1 program (which is available for the MS-DOS and VAX/VMS versions of CLiP) converts those names to upper case in the INI-file it creates. Thus the file-option arguments in the source files should definitely be specified in upper case too.

In a Unix environment a file name is case sensitive indeed. Currently no CLIP_1 exists for Unix and an INI-file has to be created by means of the short-cut routine CLP (or directly by means of an editor, cf. section 6). No conversion is performed in this case and one has to make sure the file-option argument is identical to the corresponding name in the INI-file.

7.5.2 Omitted modules

If modules are specified for being OMITTED at extraction time, it will be omitted independent of the path that may have been specified. The module will never be generated.

7.5.3 Empty run

CLiP does not always recognize a corrupted INI-file. Thus if you run CLiP and it produces a report of the form

```
============================ CLiP version 2.1 ==========================

============================ Busy scanning =============================
Scanning file: .................
.................................

============================ End scanning ==============================

============================ Busy analysing ============================

============================ End analysing =============================

============================ Busy generating ===========================

============================ End generating ============================
Used (CPU) time :5.88 Sec.
See you next time !
```

then you start checking the specifications of the target modules, since this is the most likely source of trouble. Keep in mind however, that the problem may be also arise due to a corrupted INI-file.

7.6 Lost lines

If you have stubs of the form

```
(******************  GEN_POOL global routines  ********************)
(******************************************************************)
(* routine: write_string ....                                     *)
(* ..................                                             *)
(******************************************************************)
```

you will see that the second line of the stub (the "starred" line) is not generated in the target modules. For this reason it is better to put an additional empty line in between. For example the following stub will be extracted correctly.

```
(******************  GEN_POOL global routines  *******************)

(*****************************************************************)
(* routine: write_string ....                                    *)
(* ..................                                            *)
(*****************************************************************)
```

7.7 DOS version only

If you activate the PROJECT/Load INI-file or PROJECT/Save INI-file menu of the CLIP_1 program and you specify an illegal drive in the Current Directory option, you may hang the system. For instance a drive specification preceeded by a space will produce this bug.

8 References

Ammers E.W. van, M.R. Kramer (1993), The CLiP Style of Literate Programming (submitted for publication). Anonymously ftp-able as CLIP_STYLE.PS, CLIP_STYLE_A.PS and CLIP_STYLE_B.PS from directory CLIP on sun01.info.wau.nl.

Ammers E.W. van, Kramer M.R. (1992), VAMP: A Tool for Literate Programming Independent of Programming Language and Formatter. CompEuro '92 Proceedings, May 4-8 1992, the Hague, pg. 371-376.

Knuth D.E. (1984). Literate Programming. The Computer Journal 27, 2, 97-111.

Table of Contents