THE CLE-2000 TOOL-BOX

Robert ROY
Institut de génie nucléaire
École Polytechnique de Montréal
e-mail: roy@meca.polymtl.ca

December 1999
Abstract

In reactor physics, modern computer codes use a free-format input procedure, that simplifies the tedious production of long lists of geometric and material data. This procedure will generally make a crude syntactical analysis of the reader unit file, allowing the users to call a sequential calculation but with added flexibilities such as putting data wherever they want and adding comments. This first step in defining a user-friendly input stream is still used by many engineers all over the world. Assuming a basic definition for the free-format procedure, an efficient CLE-2000 compiler can now be formally written in any language. This second step enables insertion of loops when calling modular parts of codes in order to carry out a sensitivity analysis on various steps of large codes, without recoding or defining new modules. The CLE-2000 tool-box is described in this report, along with various examples of coding in this language.
# Contents

1 Introduction 4

2 CLE-2000 syntax (user’s guide) 6
   2.1 Values and variables .................................................. 6
   2.2 Declaration statements ............................................... 8
   2.3 Operations and expressions ....................................... 9
   2.4 Executable statements ............................................... 11
   2.5 Conditional statements ............................................ 13
   2.6 Type conversion statements ...................................... 15
   2.7 Parametric constants ................................................ 15
   2.8 End of compilation .................................................. 16
   2.9 Example of plain CLE-2000 source files .......................... 17

3 CLE-2000 drivers (programmer’s guide) 19
   3.1 Routines involved in the compilation process .................... 19
      3.1.1 CLEPIL arguments ............................................. 20
   3.2 Routines involved in the execution process ..................... 21
      3.2.1 REDOPN arguments ........................................... 22
      3.2.2 REDGET arguments .......................................... 23
      3.2.3 REDPUT arguments .......................................... 25
      3.2.4 REDCLS arguments .......................................... 26
   3.3 Utility routines for CLE-2000 applications ...................... 27
      3.3.1 CLEOPN arguments .......................................... 28
      3.3.2 CLEGET arguments .......................................... 29
      3.3.3 CLEPUT arguments .......................................... 30
      3.3.4 CLECLS arguments .......................................... 30
      3.3.5 CLECOP arguments .......................................... 31
   3.4 External routine for parametric constants ....................... 32
      3.4.1 CLECST arguments .......................................... 32
3.5 Example of a simplified DRIVER ................................. 33
  3.5.1 Execution of CLE-2000 procedures ......................... 33
  3.5.2 Passing arguments between CLE-2000 procedures .......... 34

4 CLE-2000 examples (validation) 35
  4.1 Example of recursivity: 8! .................................. 35
  4.2 Values of the Bessel function $J_0(x)$ ....................... 36
  4.3 Calculations based on the Julian day ......................... 37
  4.4 Gauss-Legendre integration ................................ 38
  4.5 Finding zeros of a function ................................. 39

5 Conclusion 40

References 42

A Appendices 43
  A.1 Glossary of CLE-2000 keywords ............................ 43
  A.2 New syntactical features of version 2 ..................... 45
  A.3 Record definitions on object files ........................ 46
    A.3.1 Top-of-file record .................................. 46
    A.3.2 Input-stream records ................................ 48
    A.3.3 Variable-stack records .............................. 49
  A.4 Fortran-77 and Fortran-90 implementations ................ 50
  A.5 Listing examples ......................................... 51
  A.6 List of all compilation errors ............................ 54

Index 55
List of Figures

1. CLE-2000 compilation. ................................................. 4
2. Definition of an instruction. ........................................... 6
3. The declaration statement. ........................................... 8
4. The binary operations. ................................................ 9
5. The unary operations. ................................................ 10
6. The relational operations. .......................................... 10
7. The EVALUATE statement. .......................................... 11
8. Access from the application of the variable content. ............ 12
9. Changing the variable content in the application. ............... 12
10. The ECHO statement. ............................................... 12
11. The IF statement. ................................................ 13
12. The REPEAT statement. ........................................... 14
13. The WHILE statement. ........................................... 14
14. Type conversion operators. ....................................... 15
15. Calling a parametric constant. .................................. 15
16. Statement for the end of compilation. .......................... 16

List of Tables

1. Results returned by xmachar.c2m .................................. 18
2. Tree of routines called by CLEPIL .................................. 19
3. The package of routines controlling IO .............................. 21
4. Indirect access to the CLE-2000 stack of variables ............... 28
5. Complementary routine ............................................. 31
6. Unforgettable julian days ......................................... 38
1 Introduction

This note describes the first release of a language called CLE-2000. It provides information about the language itself and its use when embedded in applications. Defining a new programming language nowadays may sound silly; however, as will be shown in the following sections, the CLE-2000 language is so simple that a compiler can be written in any other high-level programming language. Quite similar to a calculator, the instructions in CLE-2000 are coded in a keystroke manner. To use the language does not require any particular proficiency in programming, but familiarity with a basic programming language is assumed. Like any other programming language, the first step in the CLE-2000 processing of the source file involves a compiler. The compiling step is needed in order to see if there are any syntactical errors in the source file. In this first step of the language, instructions are interpreted to see if the logic can be followed thoroughly and if types are respected within the evaluation/computation steps.

![Source file → Compiled file → Listing](image)

Figure 1: CLE-2000 compilation.

In the present version, users can code CLE-2000 instructions only in the 72 first character positions of any line of the source file. The new compiler will detect entries from columns 73 to 80, and any non-blank entry following column 72 will be considered as an error. Comments can also be included in the source file; two forms of comments are accepted by the compiler:

- an asterisk (character *) in the first column: then, the whole line is considered as a comment;

- an exclamation mark (character !) in any column of the source file: the rest of the line is considered as a comment.

Note that the old form of comments beginning with (*) and ending with *) is considered as an obsolete feature.
Except for the comment lines, blanks (characters \textbackslash {\texttt{\textbackslash \hspace{1em}}} ) are significant; they are used in order to separate variables, operations, keywords, etc. All CLE-2000 instructions (except for the last line) \textbf{must be terminated with a semicolon ; } that semicolon must also be the last instruction of that particular line of the source file. Users that do not follow these basic rules will end up with syntactical errors or unpredictable results. The CLE-2000 compiler always checks if the first word following a semicolon is a reserved keyword; forgetting a semicolon can thus have catastrophic side effects.

The CLE-2000 language provides loops, conditional testing and macro-processor capabilities. Reversed Polish Notation (RPN) was chosen for the calculator. Although it is not the conventional way to program computations in high-level languages, major benefits are that RPN helps data stacking, permits data swapping and allows computations to be processed in the exact order the user wants them done. The CLE-2000 language was kept as simple as possible; the user can use its symbolic logic, but there is no intention to compete with high-level code development. In the spirit of simplicity, the following restrictions were imposed in the actual version of CLE-2000:

- there are NO array types: should a user want to index a variable and process vectors or matrices, he has to write his application;

- there are NO implicit type coercions: too often precision is lost because users are not aware of side effects when combining single and double precision data;

- there are NO procedures or functions: should a user want to separate a procedure for subsequent use, he has to separate its source file and think of how to link it with others.

With these three restrictions, \textit{quality assurance} can be easily managed on CLE-2000 source files. The CLE-2000 compiler is responsible for producing a compiled file, where conditional logic is manageable and where computations result in data of the correct type anywhere in the source file. Once the compiler has been embedded in an application, the correct renderings of the meanings that the user wishes to express in his source file are beyond the scope of the language.
2 CLE-2000 syntax (user’s guide)

Since all CLE-2000 instructions (except for the optional QUIT statement that we will later consider) must be terminated with a semicolon ; followed by a carriage return, we will here consider that an instruction is a composed form of the following type:

![Diagram of instruction](image)

Figure 2: Definition of an instruction.

The difference between instructions and statements will become clear when we consider coupling CLE-2000 instructions with instructions of other codes or other modules. The simple fact that the semicolon may not end a CLE-2000 statement is not important to the compiler. There are two basic fields in any source file:

- CLE-2000 statements: these instructions must begin with one of the following keywords:
  
  ```
  INTEGER REAL STRING DOUBLE LOGICAL EVALUATE ECHO IF ELSEIF ELSE ENDIF WHILE ENDWHILE REPEAT UNTIL or QUIT ;
  ```

- instructions outside CLE-2000: any instructions whose first word is not in the previous list.

Instructions of these two basic fields are assumed not to overlap each other; as explained above, it is important not to forget the semicolon termination of any instruction. After each semicolon, the compiler will try to identify which instructions have a CLE-2000 meaning. For CLE-2000 statements, the semicolon will be simply withdrawn from its description.

2.1 Values and variables

Any value accepted by the CLE-2000 compiler has one of the following five types: logical, integer, real, double or string. When analyzing a possible value, CLE-2000 determines its type according
Integer any sequence of decimal numbers (unsigned integer);

Integer a sign + or − followed by any sequence of decimal numbers (signed integer);

Real a sequence of decimal numbers with one decimal point;

Real a sequence of decimal numbers, preceded or not by a sign and with or without a decimal point, with an E followed by a signed or unsigned integer;

Double sequence of decimal numbers, preceded or not by a sign and with or without a decimal point, with an D followed by a signed or unsigned integer;

String a sequence of characters enclosed between two quotation marks (character ”) or two apostrophes (character ‘) (or any sequence with no blank that was not identified as a numeric value by the previous items);

Logical a type restricted to CLE-2000 statements (as described later).

There are value limits for these different types. These limits are imposed by the language in which the CLE-2000 compiler is written. As in the original 1.0 version, the usual limits for integers, real and double precision data apply. The following restrictions are imposed on the content of a string:

• the length of any string is restricted to 72 characters;

• a string must not contain an exclamation mark (character ! reserved for comments);

• a string must not contain double IO symbols (this excludes characters << or >> whose functions will be described later);

• in CLE-2000 statements, strings must be enclosed between quotation marks (character ”);

• outside CLE-2000, strings are character sequences that were not identified as numeric (with no blank) or character sequences between apostrophes (character ’).
Variable names are restricted to 12 alphanumeric characters. Every name must begin with a letter or an underscore, the other values may be letters, digits or underscores. A particular type of variable is a pre-defined constant which begins with a $ sign (see the section on parametric constants). Once declared, these variable identifiers are uniquely associated with a memory location in the direct access (object) file. At any step in the CLE-2000 source file, the user may store a value in the variable. He may also recall the value of a variable inside any evaluation/calculation step.

### 2.2 Declaration statements

There are five types available in CLE-2000. Variables must be declared only once in a source file; they are static in the sense that, once defined, their values will stay available anywhere in the source file. The declarations can be given anywhere in the source file, as long as these declarations are done at the basic logical level (that is not inside `IF` `WHILE` or `REPEAT` statements). It is important to note that a variable must be declared before its first appearance in an executable statement. There are two forms of declarations: the simple declaration of the type and the declaration followed by initialization of the variables.

![Declaration diagram](image)

Figure 3: The declaration statement.

Using string variables, the user can manipulate strings up to 72 characters in length. Here are some examples of possible declaration:

```plaintext
LOGICAL logic_Why_not;
INTEGER i,j,k:=0,1,2;
INTEGER prime, div;
REAL a,b,c:=0.0,1.0,.02E+2;
```
The CLE-2000 Tool-box

STRING $\text{DATE} := \text{"12/21/1953"};$

DOUBLE $\text{Pi} := 3.14159265358979\times 10^0;$

In the first declaration statement, the value of the logical variable “logic” is undefined. The same thing happens in the second statement, where “prime” and “div” are not defined. However, in the four last statements, the variables are also initialized.

### 2.3 Operations and expressions

CLE-2000 uses reverse polish notation (RPN) in order to carry out arithmetic expressions. The four usual binary operations are available plus exponentiation.

![Figure 4: The binary operations.](image)

The operations are available for all the available types. So far, the language does not permit operations on different types; as stated above, this should prevent users from loosing accuracy in any operation. The user is supposed to provide two expressions of the same type. Operations with logical variables are possible. In that case, $+$ and $*$ mean OR and AND respectively. For logicals, $-$ and $/$ mean OR NOT and AND NOT. Two operations with character variables are possible: $+$ means concatenation of the two character strings and $-$ means to withdraw from the first word the ending occurrence of the second word.

Twelve of the most used unary operations were also available in CLE-2000. These basic functions will cover most of the simple mathematical tasks of a calculator. These are given in Figure 5.

These unary operations are generally available in the floating-point types REAL and DOUBLE. An exception is ABS that is also available with the INTEGER type. The NOT operation can be only used with a logical type. In Figure 6, the six relational operators used in
order to compare values are listed.

These binary comparison operations are available in order to compare two expressions of the same type. The result of these comparisons is always a logical value.

We are now able to describe expressions in the CLE-2000 language. Expressions are combinations of values and operations. The values can be either direct values of one of the five types or indirect values by recalling the contents of a CLE-2000 variable. The unary, binary and
relational operations can be combined to generate complex expressions as:

\[
b_i \times b_i \times a_i \times c_i \times 4. \times i^* - \\
\text{logic} \times \text{prime} \times \text{prime} \times \text{div} \times / \times \text{div} \times i = i + \\
fb_0. \times > \times fc_0. \times > \times fb_0. \times < \times fc_0. \times < \times i \times i + \\
s_2. \times \text{xm} \times q \times r_1 \times - \times i \times i \times i \times i \times b_i \times a_i \times - \times r_1 \times 1 \times i \times i \times i \times i \times i + \\
3. \times \text{xm} \times q \times i \times i \times toli \times q \times i \times ABS \times - \times e \times q \times i \times ABS \times >=
\]

Because plain CLE-2000 does not permit type coercion, it is possible to say that: in the first line expression, variables “a”, “b” and “c” are reals. In the second one, “logic” is a logical. In the third one, variables “fb” and “fc” are reals, etc. Note that the first expression simply calculates the value of \(b^2 - 4ac\).

Expressions can also be used in variable initialization. As long as any variable occurring in the expression has already been declared, it is possible to use it in an expression.

### 2.4 Executable statements

In CLE-2000, calculations are normally done in an **EVALUATE** instruction.

![EVALUATE](image)

Figure 7: The **EVALUATE** statement.

Here are some examples of CLE-2000 calculations:

**EVALUATE** \(i_\text{U} := i_\text{U} + 1_\text{U};\)

**EVALUATE** \(\text{logic} := \text{logic} \times \text{prime} \times \text{prime} \times \text{div} \times / \times \text{div} \times i = i +;\)

**EVALUATE** \(\text{minim} := 3. \times \text{xm} \times q \times i \times i \times toli \times q \times i \times ABS \times -;\)

When linking CLE-2000 statements with input-stream instructions of another language, access is provided to users. To access the content of a variable, the access instruction is:

Note that the access instruction is completely transparent to the non-CLE-2000 module; in other words, the module will only see the data, not the name of the variable. Users must select the appropriate type.
When developing a new module that will fully use the CLE-2000 syntax, the developer can also have a inverse access in order to put a value in a CLE-2000 variable. To put a value in a variable, the inverse access instruction is:

\[
\text{variable} \leftarrow \text{value}
\]

Figure 8: Access from the application of the variable content.

Note that the inverse access operation is done after checking if the type correctly matched the one given by the user. In order to help the developer to recover an expected value, a good programming rule would be to impose a keyword before the inverse access is taken over.

Finally, when users want to output values of CLE-2000 variables of their program, they can use the \texttt{ECHO} instruction:

\[
\text{ECHO expression}
\]

This last output statement is normally used to output values, but it can also be used to do computations as in the last of these examples:

\[
\begin{align*}
\text{ECHO prime} & \text{"is a prime number";} \\
\text{ECHO } \text{"Tolerance=} & \text{tol} \\
\text{"Number of iter=} & \text{iter} \\
\text{"Root value=} & \text{zbrent} \\
\text{ECHO } \text{"Calculation of } b^2 - 4ac \text{ gives:} & b \times a \times c \times 4 - b - c
\end{align*}
\]
2.5 Conditional statements

As a primitive programming language, CLE-2000 is able to perform loops and tests. These are the most important statements of the language because they allow repetitive calculations on data files that would normally be cumbersome to be carried out. The one-block or multi-block decision maker is similar to the one in other languages and is shown in Figure 11.

Assuming the following input:

```plaintext
IF logic THEN
    set_of_statements_1
    ECHO "This is logic...";
ELSE
    set_of_statements_2
    ECHO "This is NOT logic...";
ENDIF;
```

the conditional test on the “logic” variable determine which of the set of instructions 1 or 2 will be executed. If “logic” is true, set 1 is executed and set 2 is bypassed. If “logic” is false, set 1 is bypassed and set 2 is executed.

Plain CLE-2000 contains two kinds of conditional loops: the **REPEAT ... UNTIL** loop and the **WHILE ... ENDWHILE** loop.

In this first conditional loop
**Figure 12:** The **REPEAT** statement.

**REPEAT**

\[
\text{set of statements}
\]
\[
\text{EVALUATE } i := i \downarrow 1 + ;
\]
\[
\text{EVALUATE } \text{logic} := i \downarrow 10 = ;
\]
\[
\text{UNTIL } \text{logic} ;
\]

CLE-2000 will first do the set of instructions; then if the “logic” condition is false, CLE-2000 will repeat the whole set of instructions. If the “logic” condition ever turns out to be true, the loop will stop.

**Figure 13:** The **WHILE** statement.

**WHILE**

\[
\text{condition}
\]
\[
\text{DO}
\]
\[
\text{statements}
\]
\[
\text{ENDWHILE}
\]

In this second loop

**WHILE** \text{logic} \text{DO}

\[
\text{set of instructions}
\]
\[
\text{EVALUATE } i := i \downarrow 1 + ;
\]
\[
\text{EVALUATE } \text{logic} := i \downarrow 10 = ;
\]
\[
\text{ENDWHILE} ;
\]

the fundamental difference is that the condition is checked before entering into the loop. That means that if the “logic” condition is false, the set of instructions will not be executed at all. When using loops, one must be sure that the logical value will eventually become false; if this is not the case, an infinite loop will go on and the user will have to interrupt his execution.
2.6 Type conversion statements

As stated in the introduction, there are no implicit type coercions in CLE-2000. However, the user can change the type of a value (integer, real or double) using one of the six keywords: `R_TO_I` `D_TO_I` `I_TO_R` `D_TO_R` `I_TO_D` or `R_TO_D`. These keywords act as unary operators.

![Type conversion operators](image)

Figure 14: Type conversion operators.

The type of the expression must agree with the selected conversion (beginning with `I` for an integer expression, `R` for a real expression and `D` for a double expression). The final type is given by the last characters of the selected keyword for the chosen conversion (ending with `_I` for conversion to integer, `_R` for conversion to real and `_D` for conversion to double). An important quality-assurance feature of the CLE-2000 compiler is its ability to determine the exact type of any component of an evaluation stack.

2.7 Parametric constants

An attractive feature of this new CLE-2000 release is the possibility for developers to furnish a set of pre-defined constants. This set is available as an external function called by CLE-2000 during compilation. These parametric constants cannot be declared during compilation because their types are already imposed.

![Calling a parametric constant](image)

Figure 15: Calling a parametric constant.
Parametric constants are clearly identified by their names which have to start with a $ sign. The constant name \textit{cst\_name} can have at most 11 characters that should obey the same rules as usual variable names. Parametric constant values can be called anywhere in the CLE-2000 field of the input source file.

CLE-2000 code distribution includes an example of a routine with many basic pre-defined parametric constants (including $\pi, e, \gamma$). Some examples are:

\begin{verbatim}
$\Pi_R = 3.14159265E0 \quad \text{(REAL)}
$E_R = 2.71828183E0 \quad \text{(REAL)}
$\text{Euler\_R} = 0.577215665E0 \quad \text{(REAL)}
$\Pi_D = 3.141592653589793D0 \quad \text{(DOUBLE)}
$E_D = 2.718281828459045D0 \quad \text{(DOUBLE)}
$\text{Euler\_D} = 0.577215664901533D0 \quad \text{(DOUBLE)}
\end{verbatim}

### 2.8 End of compilation

![QUIT](image)

Figure 16: Statement for the end of compilation.

This optional statement is useful to mark the end of a CLE-2000 source file. First, we can note that this is the only statement not followed by a semicolon, but by a period (character .); moreover, all tokens of a \texttt{QUIT} statement must be on the same source line. The compiler will just discard any line in the source file following the \texttt{QUIT} command. You quit compilation at this statement which works as an end-of-file. The source file is supposed to be at the basic logical level where all \texttt{IF REPEAT} and \texttt{WHILE} statements are now closed.

There is an optional \texttt{string\_value} (that must be enclosed in quotation mark) in the \texttt{QUIT} statement. This string has two intents: firstly, it is a title that will be used on top of output files; secondly, it can also contain a few compilation and execution options that are forwarded to the CLE-2000 compiler. These options are:
• **DEBUG/NODEBUG**: if the string contains `DEBUG`, undefined and unused variables will be tracked and all statements will be appended to the output file when executing (default value is `NODEBUG`);

• **XREF/NOXREF**: if the string contains `XREF`, the compiler will append to the listing file cross references for all variables (default value is `NOXREF`);

• **LIST/NOLIST**: if the string contains `LIST`, all statements outside CLE-2000 will be appended as is to the output file when executing (default value is `NOLIST`).

In cases where you are still investigating how to perform a calculation with a new source file and/or finding some bugs in it, a good idea would be to use the option `DEBUG`. Moreover, when executing a `DEBUG` source file, every CLE-2000 statement is echoed to the output file. Because this usually means a lot of output, once the debugging process is over, you can switch back to the standard statement `QUIT`. After having compiled and checked your original source file, the `XREF` option is available to provide a cross-reference map of your variables. The `LIST` option can be useful when statements outside CLE-2000 are to be kept on the output listing. This completes the presentation of CLE-2000 statements.

### 2.9 Example of plain CLE-2000 source files

The CLE-2000 compiler can be installed in many codes to drive free-format inputs; these codes can be compiled with several compilers and on various kinds of computer. The question is now: how can we certify that CLE-2000 calculations are accurate enough? A tentative solution to this question is the plain CLE-2000 source file named `xmachar.c2m`. This routine tries to diagnose machine parameters; on typical IEEE-compliant machines, there should be no problem when executing that file. For such machines, results should be approximately equal to those given in Table 1.

Be aware that `xmachar.c2m` can produce results which are different from those given in Table 1 on nonstandard machines. Although results of CLE-2000 calculations are machine-dependent, such results are most likely portable among IEEE-compliant machines.

Another plain CLE-2000 source file is provided by the `xiclecst.c2m` file. This file simply list all the possible parametric constants included in the routine `CLECST`. 
Table 1: Results returned by `xmachar.c2m`

<table>
<thead>
<tr>
<th>precision</th>
<th>single</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibeta</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>machep</td>
<td>−23</td>
<td>−52</td>
</tr>
<tr>
<td>eps</td>
<td>$1.19 \times 10^{-7}$</td>
<td>$2.22 \times 10^{-16}$</td>
</tr>
<tr>
<td>negep</td>
<td>−24</td>
<td>−53</td>
</tr>
<tr>
<td>epsneg</td>
<td>$5.96 \times 10^{-8}$</td>
<td>$1.11 \times 10^{-16}$</td>
</tr>
<tr>
<td>iexp</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>minexp</td>
<td>−126</td>
<td>−1022</td>
</tr>
<tr>
<td>xmin</td>
<td>$1.18 \times 10^{-38}$</td>
<td>$2.23 \times 10^{-308}$</td>
</tr>
<tr>
<td>maxexp</td>
<td>128</td>
<td>1024</td>
</tr>
<tr>
<td>xmax</td>
<td>$3.40 \times 10^{38}$</td>
<td>$1.79 \times 10^{308}$</td>
</tr>
<tr>
<td>irnd</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>ngrd</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
3 CLE-2000 drivers (programmer’s guide)

In this section, we will describe the CLE-2000 compiler organization: which routines do what and so forth. In order to gain insight into how to use the CLE-2000 compiler in an application, a small driver is provided with the code distribution. This driver should help programmers to set up an environment suitable to their needs.

3.1 Routines involved in the compilation process

For any application using CLE-2000, the first step is the compilation of the source file into an object file. This work is done by the routine CLEPIL; however, the routines called by CLEPIL should not be called by the application. Here is a tree describing the calling sequence of the compiler:

Table 2: Tree of routines called by CLEPIL

<table>
<thead>
<tr>
<th>Tree of routines called by CLEPIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEPIL</td>
</tr>
<tr>
<td>CLEPIL</td>
</tr>
<tr>
<td>CLECST</td>
</tr>
<tr>
<td>CLECST</td>
</tr>
</tbody>
</table>

Now, comes a brief description of these routines:

CLEPIL  this main routine for compiling is a concatenation of 3 successive routine calls to CLELOG, CLESTK and CLEXRF. Error codes from each of these subparts are retrieved and sent back to the application.

CLECST  this external routine should be defined by the application in order to have access to specific parametric constants.

CLELOG  this routine is responsible for making a copy of the source file to the direct-access object file. Blank records and records starting with * are eliminated. Comments starting in ! column are also eliminated. Unexpected characters (with ASCII codes less than the blank one) are tracked. The consistency of strings is checked. In order to help further
steps in compiling the object file, some carriage returns are added after specific keywords \textbf{THEN}, \textbf{ELSE} \textbf{REPEAT} \textbf{DO} and after a semicolon. Then, this routine performs a first check of CLE-2000 syntax. For statements involving several keywords (\textbf{WHILE} \ldots \textbf{DO} \ldots \textbf{ENDWHILE} or \textbf{REPEAT} \ldots \textbf{UNTIL}), it will track if a statement is consistent. Moreover, some old features of CLE-2000 (version 1.0), namely the changes from the obsolete \textbf{CHARACTER} keyword to \textbf{STRING} as well as \textbf{PRINT} to \textbf{ECHO}, will be corrected directly by renaming keywords on the object file. The levels of embedded logic are numbered (user’s statements are at level 0), and the internal logic is checked by level.

\textbf{CLESTK} this routine checks the validity of variables names and the unique declaration of their type. At the end of the object file, a stack is constructed with space for all variables which are supposed undefined (this is done by writing negative values for types). The values of parametric constants are also pushed into the stack. Then, this routine checks the consistency of types for all evaluation stacks. Remaining undefined/undeclared variables are also tracked.

\textbf{CLEXRF} this routine will do cross-referencing for CLE-2000 variables in the source-file when the \textbf{XREF} option is chosen; every variable is tracked and a list of input lines where variable are used or defined is provided. When the \textbf{DEBUG} option is chosen, all undefined or unused variables are reported and a global report on warnings and errors will be provided.

Let us now describe the arguments of the public routine \textbf{CLEPIL}.

\subsection*{3.1.1 CLEPIL arguments}

The compiler \textbf{CLEPIL} is an integer function that has 4 input arguments (IN) and returns 1 output value (OUT):

\begin{verbatim}
IRETCD = CLEPIL( IREDIN, IWRITE, IUNITO, CLECST )
\end{verbatim}
### Table 3: The package of routines controlling IO

<table>
<thead>
<tr>
<th>REDOPN entry point</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDGET</td>
</tr>
<tr>
<td>REDPUT entry point</td>
</tr>
<tr>
<td>REDCLS entry point</td>
</tr>
</tbody>
</table>

The first step for any application is to call the **REDOPN** entry point for initializing the input/output process for an object file.
3.2.1 REDOPN arguments

The routine REDOPN has 3 input arguments (IN) and no output:

\[
\text{CALL REDOPN( IINP1, IOUT1, NREC )}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IINP1</td>
<td>integer</td>
<td>IN</td>
<td>unit number of the direct-access file that contains the object file to be executed (file supposed to be opened and already processed by CLEPIL)</td>
</tr>
<tr>
<td>IOUT1</td>
<td>integer</td>
<td>IN</td>
<td>unit number of the sequential formatted output file to echo comments (file supposed to be opened; if IOUT1 is set to 0, there will be no comments); these comments include: an echo of any non-CLE-2000 statement using the LIST option, and an echo of every CLE-2000 statement using the DEBUG option</td>
</tr>
<tr>
<td>NREC</td>
<td>integer</td>
<td>IN</td>
<td>record number where execution must start; this is helpful when the application wants to use several object files calling each other: then, an object file can be closed temporary at a specific record number</td>
</tr>
</tbody>
</table>

The normal value of \( NREC \) should be zero for a brand new object file that we want to execute. Any other value of \( NREC \) is supposed to be issued after a call to REDPUT, as we will see below. The routine REDOPN saves these 3 input values and these files will be used for all REDGET or REDPUT calling.
3.2.2 REDGET arguments

Then, the application will recover input data in the free-formatted REDGET routine which has no input and 5 output values:

\[
\text{CALL REDGET( ITYP, NITMA, FLOTT, TEXT, DFLOT )}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITYP</td>
<td>integer</td>
<td>OUT</td>
<td>type of the next free-formatted non-CLE-2000 word (the types are defined after this argument description)</td>
</tr>
<tr>
<td>NITMA</td>
<td>integer</td>
<td>OUT</td>
<td>integer output data when ITYP = 1; another use is: NITMA= +1 (or −1) if a true (or false) logical value was encountered when ITYP = 5</td>
</tr>
<tr>
<td>FLOTT</td>
<td>real</td>
<td>OUT</td>
<td>real output data when ITYP = 2</td>
</tr>
<tr>
<td>TEXT</td>
<td>character</td>
<td>OUT</td>
<td>string output data when ITYP = 3; another use is: TEXT is the name of the CLE-2000 variable when ITYP &lt; 0</td>
</tr>
<tr>
<td>DFLOT</td>
<td>double</td>
<td>OUT</td>
<td>double precision output data when ITYP = 4</td>
</tr>
</tbody>
</table>

The ITYP variable can take several values that describe the data mode:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITYP = −5</td>
<td>logical type (&gt;&gt; . &lt;&lt; argument),</td>
</tr>
<tr>
<td>ITYP = −4</td>
<td>double precision type (&gt;&gt; . &lt;&lt; argument),</td>
</tr>
<tr>
<td>ITYP = −3</td>
<td>string type (&gt;&gt; . &lt;&lt; argument),</td>
</tr>
<tr>
<td>ITYP = −2</td>
<td>real (single precision) type (&gt;&gt; . &lt;&lt; argument),</td>
</tr>
<tr>
<td>ITYP = −1</td>
<td>integer type (&gt;&gt; . &lt;&lt; argument),</td>
</tr>
<tr>
<td>ITYP = +1</td>
<td>integer type,</td>
</tr>
<tr>
<td>ITYP = +2</td>
<td>real (single precision) type,</td>
</tr>
<tr>
<td>ITYP = +3</td>
<td>string type,</td>
</tr>
<tr>
<td>ITYP = +4</td>
<td>double precision type,</td>
</tr>
<tr>
<td>ITYP = +5</td>
<td>logical type,</td>
</tr>
<tr>
<td>ITYP = +9</td>
<td>end-of-file was encountered on the object file,</td>
</tr>
<tr>
<td>ITYP = +10</td>
<td>input/output is closed.</td>
</tr>
</tbody>
</table>
After a call to REDOPN, an application will generally use a sequence of REDGET calls in order to get data from the CLE-2000 object file. In the routine REDGET, the only output variable that changes each time the routine is called is the ITYP variable. Depending on the type of the variable, only one of the 4 other output variables is overwritten; the value of the 3 others are protected. For the particular case of the string variable TEXT, the developer of the application is responsible for providing a sufficient length in his CHARACTER(*) declaration. Although the maximum length of a string is 72, if TEXT has a shorter length, the string will be truncated. If the next input data is the content of an undefined CLE-2000 variable, REDGET will complain by calling a routine called XABORT; there is no possible recovery from these kinds of errors. This routine is intended to print the message sent by REDGET and then to stop execution; it is the developer’s responsibility to write the XABORT routine so that he preserves important files and deletes temporary ones before closing the program. The only argument of the XABORT routine is a character string usually containing the error message. For execution of a single input source deck, the use of CLEPIL, followed by a call to REDOPN and a loop of calls to REDGET until ITYP = 9, provides a basis for driving simple programs in a free-format environment.

However, this may not be sufficient if the developer would also like to interfere with the content of CLE-2000 variables. The routine REDPUT is a kind of inverse for REDGET: it can change the value of a CLE-2000 variable during execution. If an argument is of type >> . <<, the REDGET call returns a negative ITYP value to signify the need for a new value that should be provided by the application; to help the application, the name of the variable is also given in the TEXT parameter (remember that variable names can contain up to 12 characters). Because CLE-2000 is expecting this value, the value of the TEXT variable is forced as undefined and will stay undefined until a REDPUT call is performed. These variables are kept on a LIFO stack and the REDPUT reply does not have to be immediate; however, it should be remembered that the next REDPUT call will be applied to the last argument of type >> . << encountered, and this process can go on until there are no other values to put in the stack. It is therefore the developer’s responsibility that its application keep track of the number of values to put and their types. The REDPUT routine is a developer-beware routine; the side effects produced on the stream of CLE-2000 calculations by these external inputs can be unpredictable.
### 3.2.3 REDPUT arguments

The **REDPUT** routine which has 5 input values (similar to the output values of **REDGET**): 

```fortran
CALL REDPUT( ITYP, NITMA, FLOTT, TEXT, DFLOT )
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITYP</td>
<td>integer</td>
<td>IN</td>
<td>type of the CLE-2000 variable whose value is to be changed; these types are limited to positive values: +1) integer type, +2) real (single precision) type, +3) string type, +4) double precision type, +5) logical type</td>
</tr>
<tr>
<td>NITMA</td>
<td>integer</td>
<td>IN</td>
<td>integer input data when ITYP = 1; another use is: NITMA = +1 (or −1) if a true (or false) logical value was encountered when ITYP = 5</td>
</tr>
<tr>
<td>FLOTT</td>
<td>real</td>
<td>IN</td>
<td>real input data when ITYP = 2</td>
</tr>
<tr>
<td>TEXT</td>
<td>character</td>
<td>IN</td>
<td>string input data when ITYP = 3</td>
</tr>
<tr>
<td>DFLOT</td>
<td>double</td>
<td>IN</td>
<td>double precision input data when ITYP = 4</td>
</tr>
</tbody>
</table>

When the developer calls **REDPUT**, he has to be sure that in the source file there was a command `>>variable<<`. Moreover, he also has to be sure of the type of this variable in order to correctly call the routine. There are situations where this routine may be an helpful shortcut to circumvent extensive programming.

The last useful entry point is the routine **REDCLS** that allows the developer to stop execution of an object file and still know the record where it stops. This closing process can be helpful if CLE-2000 is to be used with several files that can call one another.
3.2.4 REDCLS arguments

The routine REDCLS has 3 output arguments (similar to input arguments of REDOPN):

CALL REDCLS( IINP1, IOUT1, NREC )

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IINP1</td>
<td>integer</td>
<td>OUT</td>
<td>unit number of the direct-access object file that is currently executing</td>
</tr>
<tr>
<td>IOUT1</td>
<td>integer</td>
<td>OUT</td>
<td>unit number of the sequential formatted output file where comments are currently echoed</td>
</tr>
<tr>
<td>NREC</td>
<td>integer</td>
<td>OUT</td>
<td>record number where execution is stopped; this object file can only be reopened consistently if this record number is used as input for REDOPN</td>
</tr>
</tbody>
</table>

Using the REDGET package, even a recursive application can be driven. The only external call of this package is the routine XABORT provided by the application, and this abort feature is solely used in extreme cases of computation failures. We will now see some extensions that are provided if further interference between the application and CLE-2000 is necessary.
3.3 Utility routines for CLE-2000 applications

Some applications could also want to have an access to the content of CLE-2000 variables by their names. I call this *indirect access*. Let us look at a simple example to show the interest of this indirect access and the difference between direct and indirect access. First, if an application wants to read an integer with value 0, this CLE-2000 source file would do the job:

```plaintext
!example of direct access to CLE-2000 variables
INTEGER i := 0;
<<i>>;
```

The first `REDGET` call of the application would result with `ITYP = 1` and `NITMA = 0`; this is direct access to the content of a CLE-2000 variable.

Second, if an application wants to put the integer 0 into a variable, this would do the job:

```plaintext
!example of direct access to CLE-2000 variables
INTEGER i;
>>i<<;
```

The first `REDGET` call of the application would then result with `ITYP = -1` and `TEXT = i`; then, the application can call `REDPUT` with `ITYP = +1` and `NITMA = 0` to set the variable to 0. This is still direct access to the variable content.

Now, suppose that the CLE-2000 source file is:

```plaintext
!example of indirect access to CLE-2000 variables
INTEGER i := 0;
<<i>>;
```

The first `REDGET` call of the application would then result with `ITYP = 3` and `TEXT = 'i'`; in fact, we have read the name of a CLE-2000 variable. This does not give us direct access to the variable content (we do not even know its type). If the application expects that this string is the name a variable, it can be interesting to have utility functions to get or put values for this variable.

The indirect access is described in the following routines. However, the developer of an application should be aware that the indirect access is not forbidden, but must also be used
with care. With such indirect access, the CLE-2000 compiler would not anymore be able to track the undefined or unused variables and would be partly neutralized. As a matter of fact, the best programming for an application would imply to stay orthogonal to CLE-2000, without knowing that there are any variables at all except in some specific cases.

Table 4: Indirect access to the CLE-2000 stack of variables

| CLEOPN entry point | CLEGET       | CLEPUT entry point | CLECLS entry point |

To grant access to the variable content, the developer must use the CLEGET package that is similar to the above REDGET package. The CLEGET routine has 3 entry points (other than CLEGET itself): CLEOPN, CLEPUT and CLECLS.

3.3.1 CLEOPN arguments

First, the application has to initialize this access by calling the integer function CLEOPN whose 2 input arguments are:

```
IRETCD = CLEOPN( IINP1, IOUT1 )
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IINP1</td>
<td>integer</td>
<td>IN</td>
<td>unit number of the direct-access file that contains the object file where the REDGET package is being executed (file opened and already processed by CLEPIL)</td>
</tr>
<tr>
<td>IOUT1</td>
<td>integer</td>
<td>IN</td>
<td>unit number of the sequential formatted output file to echo errors (file opened)</td>
</tr>
<tr>
<td>IRETCD</td>
<td>integer</td>
<td>OUT</td>
<td>error code (0 if stack can be read)</td>
</tr>
</tbody>
</table>

After this CLEOPN call, the developer of an application can always use the CLEGET routine for indirect access to the variable content.
3.3.2 CLEGET arguments

The CLEGET routine is an integer function that has 1 input argument and 5 output arguments:

\[ \text{IRETCD} = \text{CLEGET}( \text{CPARM}, \text{ITYP}, \text{NITMA}, \text{FLOTT}, \text{TEXT}, \text{DFLOT}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPARM</td>
<td>character</td>
<td>IN</td>
<td>string of length 12 (CHARACTER*12) containing the name of the CLE-2000 variable</td>
</tr>
<tr>
<td>ITYP</td>
<td>integer</td>
<td>OUT</td>
<td>type of the CLE-2000 variable (if found); types are such that (0 &lt; \text{ABS}(\text{ITYP}) &lt; 6) and defined by:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(-5) logical type, but undefined value, (-4) double precision type, but undefined value, (-3) string type, but undefined value, (-2) real (single precision) type, but undefined value, (-1) integer type, but undefined value, (+1) integer type, (+2) real (single precision) type, (+3) string type, (+4) double precision type, (+5) logical type</td>
</tr>
<tr>
<td>NITMA</td>
<td>integer</td>
<td>OUT</td>
<td>integer output value when ITYP = 1; another use is: NITMA = +1 (or -1) if a true (or false) logical value was encountered when ITYP = 5</td>
</tr>
<tr>
<td>FLOTT</td>
<td>real</td>
<td>OUT</td>
<td>real output value when ITYP = 2</td>
</tr>
<tr>
<td>TEXT</td>
<td>character</td>
<td>OUT</td>
<td>string output value when ITYP = 3</td>
</tr>
<tr>
<td>DFLOT</td>
<td>double</td>
<td>OUT</td>
<td>double precision output value when ITYP = 4</td>
</tr>
<tr>
<td>IRETCD</td>
<td>integer</td>
<td>OUT</td>
<td>error code (0 if variable was found)</td>
</tr>
</tbody>
</table>

When calling CLEGET, many things can happen: the CPARM name may not exist in the stack of variable; in that case, the value of CLEGET will be 1. The CPARM name can exist (having be declared), but the value of the variable can still be undefined; in that case, the value of the type ITYP should be negative. If CPARM name exists and the variable has a defined value,
its value is returned the same way it would be done by REDGET. This indirect processing is called as a glue between the application and CLE-2000.

### 3.3.3 CLEPUT arguments

The integer function CLEPUT can be used by the application in order to store a value into a CLE-2000 variable; input arguments are similar to CLEGET:

\[
\text{IRETCD} = \text{CLEPUT} (\text{CPARM}, \text{ITYP}, \text{NITMA}, \text{FLOTT}, \text{TEXT}, \text{DFLOT})
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPARM</td>
<td>character</td>
<td>IN</td>
<td>string of length 12 (CHARACTER*12) containing the name of the CLE-2000 variable</td>
</tr>
<tr>
<td>ITYP</td>
<td>integer</td>
<td>IN</td>
<td>type of the CLE-2000 variable to be stored; types have the same definition and meaning as in CLEGET</td>
</tr>
<tr>
<td>NITMA</td>
<td>integer</td>
<td>IN</td>
<td>integer input value when ITYP = 1; another use is: \text{NITMA} = \pm 1 (or \pm 1) if a true (or false) logical value was encountered when ITYP = 5</td>
</tr>
<tr>
<td>FLOTT</td>
<td>real</td>
<td>IN</td>
<td>real input value when ITYP = 2</td>
</tr>
<tr>
<td>TEXT</td>
<td>character</td>
<td>IN</td>
<td>string input value when ITYP = 3</td>
</tr>
<tr>
<td>DFLOT</td>
<td>double</td>
<td>IN</td>
<td>double precision input value when ITYP = 4</td>
</tr>
<tr>
<td>IRETCD</td>
<td>integer</td>
<td>OUT</td>
<td>error code (0 if variable was found)</td>
</tr>
</tbody>
</table>

In the case ITYP is chosen negative, the variable becomes undefined after the CLEPUT call. Note that the absolute value of ITYP must match with the variable one.

### 3.3.4 CLECLS arguments

To complete this package, there is a routine CLECLS that can be used to stop the indirect access to the object file, arguments are:

\[
\text{IRETCD} = \text{CLECLS} (\text{IINP1}, \text{IOUT1})
\]
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IINP1</td>
<td>integer</td>
<td>OUT</td>
<td>unit number of the direct-access file that contains the object file where REDGET package is being executed (file opened and already processed by CLEPIL)</td>
</tr>
<tr>
<td>IOUT1</td>
<td>integer</td>
<td>OUT</td>
<td>unit number of the sequential formatted output file to echo errors (file opened)</td>
</tr>
<tr>
<td>IRETC</td>
<td>integer</td>
<td>OUT</td>
<td>error code (0 if stack can be read)</td>
</tr>
</tbody>
</table>

The last utility routine **CLECOP** is provided to help developers.

Table 5: Complementary routine

**CLECOP**

The routine **CLECOP** makes a copy of an object file. This can be useful when an application would like to preserve an object file for further uses or in cases of recursivity.

### 3.3.5 **CLECOP** arguments

**CLECOP** is an integer function that has 2 input arguments:

\[
\text{IRETCD} = \text{CLECOP}(\text{IUNITI, IUNIT0})
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IUNITI</td>
<td>integer</td>
<td>IN</td>
<td>unit number of the direct-access file that contains an object file to be copied (file opened and already processed by CLEPIL)</td>
</tr>
<tr>
<td>IUNIT0</td>
<td>integer</td>
<td>IN</td>
<td>unit number of the direct-access file that will contain the new copy (file opened, but its content will be destroyed by the copy process)</td>
</tr>
<tr>
<td>IRETC</td>
<td>integer</td>
<td>OUT</td>
<td>error code (0: no error found)</td>
</tr>
</tbody>
</table>
3.4 External routine for parametric constants

As stated above, the developer can define in his application an external function containing relevant parametric constants. The format of this external function is now described.

3.4.1 CLECST arguments

The CLECST routine is an integer function that has 1 input argument and 5 output arguments (with similar definitions as in CLEGET):

\[ \text{IRETCD} = \text{CLECST( CPARM, ITYP, NITMA, FLOTT, TEXT, DFLOT )} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPARM</td>
<td>character</td>
<td>IN</td>
<td>string of length 12 (CHARACTER*12) containing the name of the parametric constant (name starting with $)</td>
</tr>
<tr>
<td>ITYP</td>
<td>integer</td>
<td>OUT</td>
<td>type of the parametric constant (if found); types are such that 0 &lt; ITYP &lt; 6 and defined by: +1) integer type, +2) real (single precision) type, +3) string type, +4) double precision type, +5) logical type</td>
</tr>
<tr>
<td>NITMA</td>
<td>integer</td>
<td>OUT</td>
<td>integer output value when ITYP = 1; another use is: NITMA = +1 (or -1) if a true (or false) logical value was encountered when ITYP = 5</td>
</tr>
<tr>
<td>FLOTT</td>
<td>real</td>
<td>OUT</td>
<td>real output value when ITYP = 2</td>
</tr>
<tr>
<td>TEXT</td>
<td>character</td>
<td>OUT</td>
<td>string output value when ITYP = 3</td>
</tr>
<tr>
<td>DFLOT</td>
<td>double</td>
<td>OUT</td>
<td>double precision output value when ITYP = 4</td>
</tr>
<tr>
<td>IRETCD</td>
<td>integer</td>
<td>OUT</td>
<td>error code (0 if constant was found)</td>
</tr>
</tbody>
</table>
3.5 Example of a simplified DRIVER

The CLE-2000 source code distribution contains a simplified driver in order to present the features presented above. This driver named PL2000 has very limited options: compilation of CLE-2000 source files and execution of the produced object files with some stacking features to pass arguments between files. However, it can be used to perform an impressive set of calculations, as will be seen in the following application section.

3.5.1 Execution of CLE-2000 procedures

A declaration statement allowed in the PL2000 driver is the procedure statement:

```
PROC files
```

With this statement, you can compile CLE-2000 source files that will be needed for your main program. If the command `PROC file` has been issued by the user, three files with different extensions are involved in the compilation process:

- `file.c2m` is the name of the source file to be compiled;
- `file.o2m` is the name of the object file resulting from the compilation;
- `file.l2m` is the name of the output file used for comments and errors.

If you look in the PL2000 source file, you will see how the compiler routine CLEPIL is called to perform such a compilation. Note that the object file is be destroyed, so that it can be used in several calculations. Once the `file` has been compiled, it is possible to execute the object file using a one-argument statement:

```
EXEC file
```

Note that only one object file can be executed; it is supposed that this file has already been compiled. If the command `EXEC file` has been issued by the user, the program will look for a file named `file.o2m`, then makes a copy of it and executes the CLE-2000 instructions contained in this copy. In this simplified driver, the output of all these procedures (except the main file) has been eliminated.
3.5.2 Passing arguments between CLE-2000 procedures

The driver thus allows users to compile and execute CLE-2000 source files. However, in order to define a more extended programming environment, the driver can also be used to pass arguments between source files. Two statements can be used to interface between actual and dummy arguments. The first one is the **PUSH** statement:

```
PUSH values
```

When using **PUSH** statements, the user constructs a stack of values (content of the variables listed) to be used for subsequent calculations. The opposite of the **PUSH** statement is the **POP** statement defined by:

```
POP into-variables
```

When using **POP** statements, the user takes the values which were put into the stack by previous **PUSH** statements. The infinite stack is programmed through a direct access file that will contain values and their type; this file acts as a **LAST-IN/FIRST-OUT (LIFO)** device. For example, suppose that the user input is:

```
PUSH <<x>>, <<y>>, <<z>>;
POP >>>c, b, a;
```

The value of variable \( c \) will be the same as the value of \( z \) (also \( b = y \) and \( a = x \)). The user can put any value into the stack; however, when he gets a value, it has to be into a variable whose type exactly corresponds with the last value that was put into the stack. When looking into the **PL2000** source code, the developer can see how the package **REDGET** can be used to transfer values between source files. Using the statements contained in this simplified driver, it is possible to define explicit calculations with great similarity to other programming languages. However, you must remember that the CLE-2000 process is not optimized, so that the use of the CLE-2000 programming language for extensive calculations is not realistic.
4 CLE-2000 examples (validation)

The restrictive CLE-2000 syntax enables users not familiar with code development to perform simple computational tasks. A number of examples will now be provided in the context of the simplified driver presented in the previous section. Most of these examples are translated from Numerical Recipes.[1] It is assumed that the users are familiar with an RPN calculator.

4.1 Example of recursivity: 8!

The following procedure is contained in a file named fact.c2m. Its intent is to be used for calculation of $n!$.

```plaintext
! Example of a recursive procedure.
!
! input to "fact": *n*
! output from "fact": *n_fact*
!
INTEGER n n_fact prev_fact ;
POP >>n<< ;
IF n 1 = THEN
  EVALUATE n_fact := 1 ;
ELSE
  EVALUATE n := n 1 - ;
  PUSH <<n>> ;
  ! Here, "fact" calls itself
  EXEC fact ;
  POP >>prev_fact<< ;
  EVALUATE n_fact := n 1 + prev_fact * ;
ENDIF ;
PUSH <<n_fact>> ;
QUIT " Recursive function *fact* " .
```

This procedure shows some interesting features of the simplified driver. Note that there is no PROC fact; statement in the file fact.c2m, because we do not want the procedure call itself as a main program (when reaching the EXEC fact; statement). Thus, this procedure has to be compiled by another. The following main file named xfact.c2m will calculate 8!:

```plaintext
* Calling the recursive "fact" procedure:
*
* input to "fact": *n*
* output from "fact": *n_fact*
*
* use to compute n!
*
PROC fact ;
INTEGER n := 8 ;
PUSH <<n>> ;
```
EXEC fact ;
INTEGER n_fact ;
POP >>n_fact<< ;
ECHO "FACTORIAL: n $Bang_S '=' + n_fact ;
QUIT " Program *xfact* " .

Note the use of the parametric constant $Bang_S that contains the exclamation mark; because the exclamation mark is used as a delimiter for comments, this symbol cannot be directly put into a string. We will now show the versatility of the simplified driver for more common computing usage.

4.2 Values of the Bessel function $J_0(x)$

To understand CLE-2000 syntax, we will now present a simple example using the Bessel function $J_0(x)$. In Numerical Recipes FORTRAN V2.06h, this function is defined by:

```fortran
FUNCTION bessj0(x)
REAL bessj0,x
REAL ax,xx,z
DOUBLE PRECISION p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,r5,r6,
*s1,s2,s3,s4,s5,s6,y
SAVE p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,r5,r6,s1,s2,s3,s4,
*s5,s6
DATA p1,p2,p3,p4,p5/1.d0,-.1098628627d-2,.2734510407d-4,
*-.2073370639D-5,.2093887211D-6/, q1,q2,q3,q4,q5/
*-.1562499995D-1,.1430488765D-3,-.6911147661d-5,.7621095161d-6,-.934945152d-7/
DATA r1,r2,r3,r4,r5,r6/57568490574.d0,-13362590354.d0,
*651619650.7d0,-11214424.18d0,77392.3301d0,-184.9052456d0/,s1,s2,
*s3,s4,s5,s6/57568490411.d0,1029532985.d0,9494680.718d0,
*59272.6485360,267.8532712d0,1.d0/
if(abs(x).lt.8.)then
  y=x**2
  bessj0=(r1+y*(r2+y*(r3+y*(r4+y*(r5+y*r6)))))/(s1+y*(s2+y*(s3+y*
*s4+y*(s5+y*s6))))
else
  ax=abs(x)
  z=6./ax
  y=z**2
  xx=ax-.785398164
  bessj0=sqrt(.636619772/ax)*(cos(xx)*(p1+y*(p2+y*(p3+y*(p4+y*
*p5))))-z*sin(xx)*(q1+y*(q2+y*(q3+y*(q4+y*q5)))))
endif
return
END
```

Translation of the same function in CLE-2000 would look like (using a dummy input argument $x$):

```cle-2000
REAL bessj0 x ;
REAL ax xx z ;
DOUBLE p1 p2 p3 p4 p5 := 1.0d0 -.1098628627d-2 .2734510407d-4
-.2073370639D-5 .2093887211D-6 ;
DOUBLE q1 q2 q3 q4 q5 := -.1562499995d-1 .1430488765D-3 -.6911147661d-5
.7621095161d-6 -.934945152d-7 ;
```
DOUBLE r1 r2 r3 r4 r5 r6 := 57568490574.D0 -13362590354.D0
651619640.7D0 -11214424.18D0 77392.33017D0 -184.9052456D0 ;
DOUBLE s1 s2 s3 s4 s5 s6 := 57568490411.D0 1029532985.D0
9494680.718D0 59272.64853D0 267.8532712D0 1.D0 ;
DOUBLE y ;
POP >>x<< ;
IF x ABS 8. < THEN
EVALUATE y := x x * R_TO_D ;
EVALUATE bessj0 := r6 y * r5 + y * r4 + y * r3 + y * r2 + y * r1 +
s6 y * s5 + y * s4 + y * s3 + y * s2 + y * s1 + /
D_TO_R ;
ELSE
EVALUATE ax := x ABS ;
EVALUATE z xx := 8. ax / ax .785398164 - ;
EVALUATE y := z z * R_TO_D ;
EVALUATE bessj0 := p5 y * p4 + y * p3 + y * p2 + y * p1 +
xz COS R_TO_D *
q6 y * q4 + y * q3 + y * q2 + y * q1 +
xz SIN z * R_TO_D * -
.636619772 ax / SQRT R_TO_D *
D_TO_R ;
ENDIF ;
PUSH <<bessj0>> ;
QUIT " Function bessj0 " .

This function, named bessj0.c2m given with the CLE-2000 source files distribution, can be tested using the program xbessj0.c2m. This program is just a loop for calling \( J_0(x) \) for \( x \) from \(-5.0\) to \(+15.0\) incremented in steps of 1. Calculations done in the CLE-2000 routine are equivalent to those done in the FORTRAN function; all results should therefore agree.

### 4.3 Calculations based on the Julian day

You can use the program xjulian.c2m in order to compute the Julian days corresponding to a set of given dates. It is an transcription of a similar example taken from Numerical Recipes. Results are given in Table 6.

The julian day routine can also be used in conjunction with the routine flmoon.c2m (which computes the phases of the moon) in order to find every occurrence of Friday the 13th when the moon is full. This is the program badluk.c2m in which we try to find such occurrences between 1970 and 2000. Using the Eastern Standard Time (GMT-5), there are only three such cases: 11/13/1970 at 2:00, 2/13/1987 at 16:00 (was it visible at all?) and 10/13/2000 at 4:00.
Table 6: Unforgettable julian days

<table>
<thead>
<tr>
<th>Month</th>
<th>Day</th>
<th>Year</th>
<th>Julian Day</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>December</td>
<td>31</td>
<td>-1</td>
<td>1721423</td>
<td>End of millennium</td>
</tr>
<tr>
<td>January</td>
<td>1</td>
<td>1</td>
<td>1721424</td>
<td>One day later</td>
</tr>
<tr>
<td>October</td>
<td>14</td>
<td>1582</td>
<td>2299170</td>
<td>Day before Gregorian calendar</td>
</tr>
<tr>
<td>October</td>
<td>15</td>
<td>1582</td>
<td>2299161</td>
<td>Gregorian calendar adopted</td>
</tr>
<tr>
<td>January</td>
<td>17</td>
<td>1706</td>
<td>2344180</td>
<td>Benjamin Franklin born</td>
</tr>
<tr>
<td>April</td>
<td>14</td>
<td>1865</td>
<td>2402341</td>
<td>Abraham Lincoln shot</td>
</tr>
<tr>
<td>April</td>
<td>18</td>
<td>1906</td>
<td>2417319</td>
<td>San Francisco earthquake</td>
</tr>
<tr>
<td>May</td>
<td>7</td>
<td>1915</td>
<td>2420625</td>
<td>Sinking of the Lusitania</td>
</tr>
<tr>
<td>July</td>
<td>20</td>
<td>1923</td>
<td>2423621</td>
<td>Pancho Villa assassinated</td>
</tr>
<tr>
<td>May</td>
<td>23</td>
<td>1934</td>
<td>2427581</td>
<td>Bonnie and Clyde eliminated</td>
</tr>
<tr>
<td>July</td>
<td>22</td>
<td>1934</td>
<td>2427641</td>
<td>John Dillinger shot</td>
</tr>
<tr>
<td>April</td>
<td>3</td>
<td>1936</td>
<td>2428262</td>
<td>Bruno Hauptman electrocuted</td>
</tr>
<tr>
<td>May</td>
<td>6</td>
<td>1937</td>
<td>2428660</td>
<td>Hindenburg disaster</td>
</tr>
<tr>
<td>July</td>
<td>26</td>
<td>1956</td>
<td>2435681</td>
<td>Sinking of the Andrea Doria</td>
</tr>
<tr>
<td>June</td>
<td>5</td>
<td>1976</td>
<td>2442935</td>
<td>Teton dam collapse</td>
</tr>
<tr>
<td>May</td>
<td>23</td>
<td>1968</td>
<td>2440000</td>
<td>Julian Day 2440000</td>
</tr>
</tbody>
</table>

4.4 Gauss-Legendre integration

The routine \texttt{gauleg.c2m} computes the abscissas and weights for the Gaussian formula

\[
\int_{x_1}^{x_2} f(x) dx = \sum_{j=1}^{N} w_j f(x_j)
\]

This routine can be checked using the main program \texttt{xgauleg.c2m} which computes for \(N = 10\) the abscissas and weights, and then performs the numerical integration of

\[
I = \int_{0}^{10} xe^{-x} dx
\]
4.5 Finding zeros of a function

In this last example, we use the Van Wijngaarden-Dekker-Brent method for finding zeros of a function; this method has been included as a standard numerical recipe (here called zbrent.c2m) and can be programmed in any high-level language. The method is a combination of the bisection, root bracketing and inverse quadratic interpolation. The main program, called xzbrent.c2m, will compute roots of the Bessel $J_0$ function. This program works in two steps: first there is an inward search for brackets on roots that is performed by the routine zbrak.c2m, then each root is found using its specific bracket interval. On the interval $[1,50]$, sixteen roots of $J_0(x) = 0$ are found.
5 Conclusion

The language CLE-2000 has no use by itself. However, its use when embedded into large codes can lead to a significant increase in functionality for a great many applications. The whole idea is to pre-process the input (reader) file in order to make dynamic calls to routines that would be normally accessed only sequentially. You will find the first application of CLE-2000 in the generalized driver, originally devised to use object-oriented programming.[2] This generalized driver has been used to drive the two main codes used at the Institute:

DRAGON a collection of modules that includes all the functions of a lattice-cell code: resonance self-shielding and multidimensional transport calculations;[3]

DONJON a collection of modules for static and kinetics reactor core calculations.[4]

Several aspects of the language CLE-2000 currently available in the generalized driver have been presented. The fact that parametric studies can be done consistently with the framework of the original sequential code is of particular interest in the context of defining an integrated reactor model using control algorithms. In the coming years, it is expected that all our codes may benefit from the dynamic access to the input file as generated with CLE-2000. This way of doing things should also improve our computations in the sense that without doing any file editing, we can generate a whole library of nuclear properties and carry out sensitivity analysis. This approach will permit the treatment of complex simulations and the debugging of modular parts of our codes will be easier. The modular calculation strategy will also ensure that subsequent developments can be easily implemented in a fully-integrated computational environment. The fact that students can learn and be involved in developing codes in a more user-friendly environment is also an important benefit.

The new CLE-2000 standard has been defined in such a way that, at compile time, most of the constraints of the language can be verified. The new compiler is able to report the reasons for rejecting a source file and, provided that the machines are IEEE-compliant, the source files producing correct results should also be portable.
I will end this conclusion by listing some crucial arguments that could help to convince people to use the CLE-2000 compiler:

1. the consistent treatment of types for variables is more severe than in most high-level modern programming languages;

2. the exact order in which operations are performed is strictly determined by the user’s input;

3. the undefined/unused variables can be tracked and suppressed to improve user’s template source files and to insure that such input files contain only what was intended to be computed;

4. the static memory required by CLE-2000 computations when executing an object file is minimal (in fact, less than 8KB are needed), so that an application’s developer does not have to care about it;

5. the compiler itself is so small (the sum of all coding lines is about 4K including comments) that its quality assurance can be easily managed over the coming years.
Acknowledgements

This work has been carried out partly with the help of grants from Hydro-Québec and the Natural Sciences and Engineering Research Council of Canada. I would like to thank my two β-testers, Elisabeth Varin and Siamak Kaveh; they have greatly helped me in clarifying error messages issued by the compiler. I would also like to thank Peter Tye for his help in reviewing the documentation.

References


A Appendices

A.1 Glossary of CLE-2000 keywords

In this version of the CLE-2000 language, the following 36 keywords are used:

**ABS**  Unary operator, returns the absolute value

**ARCCOS**  Unary operator, returns the arc-cosine in radians

**ARCSIN**  Unary operator, returns the arcsine in radians

**ARCTAN**  Unary operator, returns the arctangent in radians

**CHS**  Unary operator, changes the sign of any value

**COS**  Unary operator, returns cosine of radians

**D_TO_I**  Type conversion from double to integer

**D_TO_R**  Type conversion from double to real

**DO**  Mandatory keyword after the condition of a WHILE statement

**DOUBLE**  Starts declaration of double precision variables

**ECHO**  Starts printing to the output file

**ELSE**  Where it goes when the IF and all ELSEIF conditions are not met

**ELSEIF**  Optional conditional

**ENDIF**  Closes the IF/ELSEIF/ELSE conditional sequence

**ENDWHILE**  Closes the WHILE/DO sequence

**EXP**  Unary operator, returns the exponential

**EVALUATE**  Executable statement used for calculations

**I_TO_D**  Type conversion from integer to double

**I_TO_R**  Type conversion from integer to real
**IF**  Starts a conditional statement

**INTEGER**  Starts declaration of integer variables

**LOGICAL**  Starts declaration of logical variables

**LN**  Unary operator, returns the natural logarithm

**NOT**  Unary operator, returns the logical negation

**QUIT**  End compilation there

**R_TO_D**  Type conversion from real to double

**R_TO_I**  Type conversion from real to integer

**REAL**  Starts declaration of real variables

**REPEAT**  Starts a REPEAT/UNTIL loop

**SIN**  Unary operator, returns the sine of radians

**SQRT**  Unary operator, returns the square root

**STRING**  Starts declaration of string variables

**TAN**  Unary operator, returns the tangent of radians

**THEN**  Mandatory keyword after the condition of a IF or ELSEIF statement

**UNTIL**  Closes the REPEAT loop

**WHILE**  Starts a WHILE/DO/ENDWHILE loop
A.2 New syntactical features of version 2

Differences with version 1 of CLE-2000 are:

1. The old form of comments (* ... *) was replaced by ! comments.

2. The keyword CHARACTER was replaced by STRING.

3. The user-defined strings MUST now be enclosed in double quotes "".

4. The keyword PRINT was replaced by ECHO.

5. The keyword TRUE and FALSE are replaced by $True_L and $False_L.

6. Extension of conditional statements with ELSEIF.

7. New unary operation: SQRT.

8. New binary operation: **.

9. New type conversion operators I_TO_R, I_TO_D, R_TO_D, R_TO_I, D_TO_I, D_TO_R.

10. New ending statements: QUIT.

11. New capability to define parametric constants.

The compilation process now tries to analyze the source file as much as possible to detect internal CLE-2000 hazards (or errors). One of the main improvement in version 2 is that all non-CLE-2000 statements can now be echoed (using the LIST option) in the output listing.
A.3 Record definitions on object files

The direct-access object file is an executable file in the CLE-2000 system. In this appendix, the structure of records after compilation is described. There are 3 types of records: the top-of-file record, the input-stream records and the variable-stack records.

The first record (REC=1) contains the main values needed to extract information from the object file. The arrangement of further records is: from (REC=2) to (REC=NINPUT), the image of the source file; from (REC=NINPUT+1) to (REC=NINPUT+NSTACK), the variable-stack records sorted by ascending order of names.

In order to store all these records, the record length must normally be at least 120 bytes (exactly on most 32-bit machines). On some systems, it may be better to allocate a multiple of 2, so that 128 would then be an appropriate record length. Even assuming a regular 32-bit machine (with 1-byte characters, 4-byte integers and 8-byte doubles), the records do not have exactly the same length: there are 2 integer slots left in (REC=1) and 1 integer slot left in variable-stack records. These slots can be used if needed.

An example of that may be the case where a developer would like to store supplementary data in the object file. If the developer of an application adds records for his own use at the end-of-file (and this may be a good idea), the top-of-file record parameters needs to be changed (or some may be added). The CLECOP routine must also be upgraded if this routine is to be used for also copying these new records. The best way to code such applications is to first pass through the CLEPIL process and only then write other records at the end (after REC=NINPUT+NSTACK).

A.3.1 Top-of-file record

The first record (REC=1) contains:

\texttt{CL2000,MYTITL,NRECOR,NINPUT,MAXLVL,NSTACK,IXRLST,IOULST,IDBLST}
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL2000</td>
<td>character</td>
<td>string of length 12 (CHARACTER*12) containing (used for validation of object file, exact content is &quot;CLE2000(V21)&quot; )</td>
</tr>
<tr>
<td>MYTITL</td>
<td>character</td>
<td>string of length 72 (CHARACTER*72) containing the title and options given in the QUIT statement (blanks when no ending statement)</td>
</tr>
<tr>
<td>NRECOR</td>
<td>integer</td>
<td>total number of records in the object file (it has to be changed if an application needs to use the heap of the object file)</td>
</tr>
<tr>
<td>NINPUT</td>
<td>integer</td>
<td>number of records used for statements (pre-processed copy of the input-stream records from the source file) in the object file plus one (myself as the top-of-file record)</td>
</tr>
<tr>
<td>MAXLVL</td>
<td>integer</td>
<td>maximum number of logical levels used in the object file</td>
</tr>
<tr>
<td>NSTACK</td>
<td>integer</td>
<td>number of records used for keeping variable names and contents (called variable-stack records)</td>
</tr>
<tr>
<td>IXRLST</td>
<td>integer</td>
<td>number controlling the access to the cross-reference in compiling mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((-1) means no cross referencing of the variables; (+1) means cross referencing of the variables on the output unit)</td>
</tr>
<tr>
<td>IOULST</td>
<td>integer</td>
<td>number controlling the access to the output listing mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((-1) means no output for non-CLE-2000 statements; (+1) means that all non-CLE-2000 statements are listed on the output unit)</td>
</tr>
<tr>
<td>IDBLST</td>
<td>integer</td>
<td>number controlling the access to the debugger in executing mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((-1) means no debugging on the output unit; (+1) means debugging and output of all statements on the output unit)</td>
</tr>
</tbody>
</table>

The values of CL2000, NINPUT, MAXLVL and NSTACK must not be changed by any application.

At the end of the compilation process, we have that \( \text{NRECOR} = \text{NINPUT} + \text{NSTACK} \).
A.3.2 Input-stream records

From REC=2 to REC=NINPUT, the input-stream records (image of the user’s source file) have the following description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPARIN</td>
<td>character</td>
<td>string of length 12 (CHARACTER*12) containing the names for CLE-2000 statements or blanks</td>
</tr>
<tr>
<td>MYRECO</td>
<td>character</td>
<td>string of length 72 (CHARACTER*72) containing the image of a record that was read from the source file (all-blank and all-comment records are withdrawn)</td>
</tr>
<tr>
<td>ILINES</td>
<td>integer</td>
<td>corresponding line in the input source deck</td>
</tr>
<tr>
<td>ILEVEL</td>
<td>integer</td>
<td>logical level of this record (statements outside CLE-2000 are at 0-level, declarations are at 1-level, conditional statements are at higher level depending on the number of previous conditions)</td>
</tr>
<tr>
<td>JRECOR</td>
<td>integer</td>
<td>branching record for conditional statements (this value must range between 2 and NINPUT)</td>
</tr>
<tr>
<td>MASKCK</td>
<td>integer</td>
<td>vector of 3 integers containing a mask of the record</td>
</tr>
<tr>
<td>IPACKI</td>
<td>integer</td>
<td>vector of 3 integers containing the list of types for words in the record</td>
</tr>
</tbody>
</table>

The input-stream records have a greater length than the top-of-file record. None of these values should be changed.
A.3.3 Variable-stack records

From REC=NINPUT+1 to REC=NINPUT+NSTACK, the variable-stack records have the following description:

*CPARAV*, *CDATAV*, *INDLEC*, *IDATAV*, *ADAV*, *DDAV*, *IDCLIN*, *IDEFIN*, *IUSEIN*

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPARAV</td>
<td>character</td>
<td>string of length 12 (CHARACTER*12) containing the name of a CLE-2000 variable (or a parametric constant)</td>
</tr>
<tr>
<td>CDATAV</td>
<td>character</td>
<td>string of length 72 (CHARACTER*72) containing the value of a CLE-2000 string variable when INDLEC = 3</td>
</tr>
<tr>
<td>INDLEC</td>
<td>integer</td>
<td>integer that gives the kind of this (still undefined) variable, INDLEC&lt;0 and kind -INDLEC is 1 for integer, 2 for real, etc. INDLEC is positive for parametric constants and will become positive when executing the object file</td>
</tr>
<tr>
<td>IDATAV</td>
<td>integer</td>
<td>integer data that gives the value of a CLE-2000 integer (INDLEC = 1) or logical (INDLEC = 5 with −1:false, +1:true) variable; when INDLEC = 3, then IDATAV gives the length for the string contained in CDATAV</td>
</tr>
<tr>
<td>ADAV</td>
<td>real</td>
<td>real data that gives the value of a CLE-2000 real variable when INDLEC = 2</td>
</tr>
<tr>
<td>DDAV</td>
<td>double</td>
<td>double precision data that gives the value of a CLE-2000 double variable when INDLEC = 4</td>
</tr>
<tr>
<td>IDCLIN</td>
<td>integer</td>
<td>line number in the input source file where the variable has been declared</td>
</tr>
<tr>
<td>IDEFIN</td>
<td>integer</td>
<td>line number in the input source file where the variable was first defined (where a value is given to it)</td>
</tr>
<tr>
<td>IUSEIN</td>
<td>integer</td>
<td>line number in the input source file where the variable was first needed for calculation or output</td>
</tr>
</tbody>
</table>

The *IDCLIN*, *IDEFIN* and *IUSEIN* values are no longer used after compilation.
A.4 Fortran-77 and Fortran-90 implementations

The FORTRAN-77 version is straightforward and standard. The record length can be different on some machines, so that you may have to correct some file parameters in the main PL2000.f. You normally just need to compile the 11 routines:

CLECOP.f
CLECST.f
CLEGET.f
CLELOG.f
CLEPIL.f
CLESTK.f
CLEXRF.f
PL2000.f
PL2CPU.f
REDGET.f
XABORT.f

The FORTRAN-90 implementation is standard and fully portable. It contains a certain number of modules and routines. The following order of compilation will ensure that USE statement are correctly sequenced:

C2MTYP.f
C2MPIL.f
C2MRED.f
C2MVAR.f
CLECOP.f
CLECST.f
PL2CPU.f
PL2000.f
XABORT.f

The module C2MTYP.f contains general type definitions, all keywords of the CLE-2000 language and some constant parameters. The module C2MPIL.f contains the public routine CLEPIL. The module C2MRED.f contains the four public routines: REDOPN REDGET REDPUT and REDCLS. The module C2MVAR.f contains the four public routines: CLEOPN CLEGET CLEPUT and CLECLS.

Note that the main routine PL2000.f is the simplified driver. The routine PL2CPU.f contains an implementation for calculating the CPU time in FORTRAN-90 (it returns 0.0 in FORTRAN-77).
A.5  Listing examples

Let us consider the following source file (using some obsolete version-1 features):

\[
\begin{align*}
(* \text{this example will compute odd prime numbers up to 100} *)
\end{align*}
\]

\[
\begin{align*}
&\text{LOGICAL } lprime ; \\
&\text{INTEGER } \text{div } pmax \text{ prime } := 0 \ 99 \ 3 ; \\
&\text{PRINT prime 'is a prime number' ;} \\
&\text{REPEAT} \\
&\text{EVALUATE prime } := \text{prime } 2 + ; \\
&\text{EVALUATE div } lprime := 1 \ \text{FALSE} ; \\
&\text{REPEAT} \\
&\text{EVALUATE div } := \text{div } 2 + ; \\
&\text{EVALUATE lprime } := \text{lprime prime div div } * = + ; \\
&\text{UNTIL div div * prime }> ; \\
&\text{IF lprime } \text{NOT THEN} \\
&\text{PRINT prime 'is a prime number' ;} \\
&\text{ENDIF} ; \\
&\text{UNTIL prime pmax }> = ;
\end{align*}
\]

Calling the \textsc{CLEPIL} routine will produce the following listing:

```
* CLE-2000 VERS 2.1 * R.ROY, EPM COPYRIGHT 1999 * LINE
(* this example will compute odd prime numbers up to 100 *)
?? ??
! WARNING: (* ... *) OBSOLETE COMMENTS (USE ! INSTEAD)
LOGICAL lprime ;
INTEGER div pmax prime := 0 99 3 ;
PRINT prime 'is a prime number' ;
! WARNING: *PRINT* => *ECHO* (REPLACED)
REPEAT
EVALUATE prime := prime 2 + ;
EVALUATE div lprime := 1 FALSE ;
REPEAT
EVALUATE div := div 2 + ;
EVALUATE lprime := lprime prime prime div / div * = + ;
UNTIL div div * prime > ;
IF lprime NOT THEN
PRINT prime 'is a prime number' ;
ENDIF ;
UNTIL prime pmax >= ;
QUIT .
```

Some warnings are given by the CLE-2000 compiler, but only one error was found: the obsolete use of \textsc{FALSE} (no longer a keyword). Now, let us correct this mistake by defining the variable \textsc{FALSE}. Then, compilation ends with no error, so execution will produce the following listing:
* PL2000: COMPIling _M ain.c2 File
* CLE-2000 VERS 2.1 * R.ROY, EPM COPYRIGHT 1999 * LINE
LOGICAL FALSE := $False_L ; 0001
(\* this example will compute odd prime numbers up to 100 \*) 0002
\* WARNING: (\* ... \*) OBSOLETE COMMENTS (USE \! INSTEAD)
LOGICAL lprime ; 0003
INTEGER div pmax prime := 0 99 3 ; 0004
PRINT prime 'is a prime number' ; 0005
\* WARNING: \*PRINT* \! \! \*ECHO* (REPLACED)
REPEAT 0006
EVALUATE prime := prime 2 + ; 0007
EVALUATE div lprime := 1 FALSE ; 0008
REPEAT 0009
EVALUATE div := div 2 + ; 0010
EVALUATE lprime := lprime prime prime div / div * = + ; 0011
UNTIL div div * prime > ; 0012
IF lprime NOT THEN 0013
PRINT prime 'is a prime number' ; 0014
\* WARNING: \*PRINT* \! \! \*ECHO* (REPLACED)
ENDIF ; 0015
UNTIL prime pmax >= ; 0016
QUIT . IMPLICIT
* PL2000: EXECUTING _M ain.o2 File

>\|3 is a prime number \>|0005
>\|5 is a prime number \>|0014
>\|7 is a prime number \>|0014
>\|11 is a prime number \>|0014
>\|13 is a prime number \>|0014
>\|17 is a prime number \>|0014
>\|19 is a prime number \>|0014
>\|23 is a prime number \>|0014
>\|29 is a prime number \>|0014
>\|31 is a prime number \>|0014
>\|37 is a prime number \>|0014
>\|41 is a prime number \>|0014
>\|43 is a prime number \>|0014
>\|47 is a prime number \>|0014
>\|53 is a prime number \>|0014
>\|59 is a prime number \>|0014
>\|61 is a prime number \>|0014
>\|67 is a prime number \>|0014
>\|71 is a prime number \>|0014
>\|73 is a prime number \>|0014
>\|79 is a prime number \>|0014
>\|83 is a prime number \>|0014
>\|89 is a prime number \>|0014
>\|97 is a prime number \>|0014

* PL2000: EXECUTION HAS ENDED WELL.
* PL2000: TIMING \! \! 0.22 SECONDS.
* PL2000: END.

However, suppressing obsolete features and adding the XREF option produces a nicer listing:
This example will compute odd prime numbers up to 100,000.

```plaintext
LOGICAL lprime;
INTEGER div, pmax, prime := 0;
ECHO prime "is a prime number";
REPEAT
  EVALUATE prime := prime + 2;
  EVALUATE div lprime := 1 $False_L;
  REPEAT
    EVALUATE div := div + 2;
    EVALUATE lprime := lprime prime prime div / div * = +;
  UNTIL div div * prime >;
  IF lprime NOT THEN
    ECHO prime "is a prime number";
  ENDIF;
UNTIL prime pmax >=;
QUIT "program *prime* NODEBUG/XREF/NOLIST ".
```

Variable cross reference:

- **$False_L** found in lines 0007
- **div** found in lines 0003, 0007, 0009, 0010, 0011
- **lprime** found in lines 0002, 0007, 0010, 0012
- **pmax** found in lines 0003, 0003
- **prime** found in lines 0003, 0004, 0006, 0009, 0010, 0011, 0013, 0015

Output:

```
> 3 is a prime number
> 5 is a prime number
> 7 is a prime number
> 11 is a prime number
> 13 is a prime number
> 17 is a prime number
> 19 is a prime number
> 23 is a prime number
> 29 is a prime number
> 31 is a prime number
> 37 is a prime number
> 41 is a prime number
> 43 is a prime number
> 47 is a prime number
> 53 is a prime number
> 59 is a prime number
> 61 is a prime number
> 67 is a prime number
> 71 is a prime number
> 73 is a prime number
> 79 is a prime number
> 83 is a prime number
> 89 is a prime number
> 97 is a prime number
```

Program executed well in 0.24 seconds.
A.6 List of all compilation errors

Warnings and errors reported by CLELOG, CLESTK and CLEXRF:

! WARNING: ( ... ) OBSOLETE COMMENTS (USE ! INSTEAD)
! WARNING: *CHARACTER* => *STRING* (REPLACED)
! WARNING: *PRINT* => *ECHO* (REPLACED)
! WARNING: OUTSIDE CLE-2000, ENCLOSE STRINGS IN ’...’ (REPLACED)
! WARNING: INSIDE CLE-2000, ENCLOSE STRINGS IN “...” (REPLACED)

! CLELOG: UNEXPECTED CHARACTERS REPLACED WITH BLANKS
! CLELOG: UNBALANCED OPENING OR CLOSING STRINGS
! CLELOG: MISPLACED SEMICOLON ...; OR ;... OR ...;
! CLELOG: CHARACTERS SUPPRESSED OUTSIDE COLUMN RANGE 1:72
! CLELOG: << AND >> NOT ALLOWED IN STRINGS (SUPPRESSED)
! CLELOG: QUIT ”...”. SHOULD APPEAR ALONE A SINGLE LINE
! CLELOG: INVALID 1-CHARACTER WORD IN CLE-2000
! CLELOG: MORE THAN 12-CHARACTER WORD IN CLE-2000
! CLELOG: KEYWORD= *...*, BUT MAXIMUM NUMBER OF LEVELS IS ACHIEVED
! CLELOG: REWRITE YOUR LOGIC
! CLELOG: AFTER *...*, NOT EXPECTING KEYWORD= *...*
! CLELOG: KEYWORD= *...*, BUT NOTHING LEFT FOR THIS LEVEL
! CLELOG: KEYWORD= *...*, BUT THE NUMBER OF EQUALS *=*= IS ...
! CLELOG: KEYWORD= *...*, BUT THE NUMBER OF WORDS IS ...
! CLELOG: DECLARATION AS *...* MUST APPEAR AT LOGIC LEVEL 1
! CLELOG: INCONSISTENT END-OF-FILE, LOGIC LEVEL IS ...> 1
! CLELOG: EXPECTING *...* AT THE END OF STATEMENT *...*
! CLELOG: WRITING RETURN CODE =....
! CLELOG: READING RETURN CODE =....

! CLESTK: INVALID VARIABLE NAME IN RECORD
! CLESTK: *...* CANNOT BE DECLARED
! CLESTK: VARIABLE DECLARED TWICE *...*’
! CLESTK: VARIABLE NOT YET DECLARED *...*
! CLESTK: INVALID PARAMETER *',CPARAV,'*
! CLESTK: INVALID VARIABLE FOR >>.<< OR <<.>>
! CLESTK: INVALID TYPE_TO_TYPE CONVERSION
! CLEXRF: IOSTAT RETURN CODE =....
! CLEXRF: IMPOSSIBLE TO USE THIS *OBJECT* FILE
! CLEXRF: IMPOSSIBLE TO USE OLD *OBJECT* FILE
Index

Keyword

ABS, 9, 10, 43
ARCCOS, 10, 43
ARCSIN, 10, 43
ARCTAN, 10, 43
CHS, 10, 43
COS, 10, 43
D_TO_I, 15, 43, 45
D_TO_R, 15, 43, 45
DO, 14, 20, 43
DOUBLE, 6, 8, 9, 43
ECHO, 6, 12, 20, 43, 45
ELSE, 6, 13, 20, 43
ELSEIF, 6, 13, 43, 45
ENDIF, 6, 13, 43
ENDWHILE, 6, 13, 14, 20, 43
EVALUATE, 6, 11, 43
EXP, 10, 43
I_TO_D, 15, 43, 45
I_TO_R, 15, 43, 45
IF, 6, 8, 13, 16, 44
INTEGER, 6, 8, 9, 44
LN, 10, 44
LOGICAL, 6, 8, 44
NOT, 9, 10, 44
QUIT , 6, 16, 17, 44, 45, 47
R_TO_D, 15, 44, 45
R_TO_I, 15, 44, 45
REAL, 6, 8, 9, 44
REPEAT, 6, 8, 13, 14, 16, 20, 44
SIN, 10, 44
SQRT, 10, 44, 45
STRING, 6, 8, 20, 44, 45
TAN, 10, 44
THEN, 13, 20, 44
UNTIL, 6, 13, 14, 20, 44
WHILE, 6, 8, 13, 14, 16, 20, 44

Routine

CLECLS, 28, 30, 50
CLECOP, 31, 46
CLECST, 17, 19
CLEGET, 28–30, 32, 50
CLELOG, 19, 54
CLEOPN, 28, 50
CLEPIL, 19, 20, 24, 33, 46, 50, 51
CLEPUT, 28, 30, 50
CLESTK, 19, 20, 54
CLEXRF, 19, 20, 54
REDCLS, 21, 25, 26, 50
REDGET, 21–28, 30, 31, 34, 50
REDOPN, 21, 22, 24, 26, 50
REDPUT, 21, 22, 24, 25, 27, 50
XABORT, 24, 26