The Interpreter Clif
User's Guide

Ľ. Koreň

For version 0.8.2

Last updated 30 July 1996

Chapter 1

Preface

The syntax of the Clif is based on the C language. The handbook is sequenced very close to the [10] or other computer programming language books.

The syntax structures are described in the following chapters. These structures are described using examples. All examples are simply stated programs and therefore they should be understandable for users knowing a C-language.

Contents

List of Figures

Chapter 2

Introduction to the interpreter Clif

Our goal is to show main structures of the language via examples. We want to explain all structures in the way of writing simply stated programs.

2.1 Start of the interpreter

The interpreter session can begin by writing just Clif on the command interpreter line. The interpreter framework is started (another way of starting the interpreter session is explained later). On the standard output is written:

```
This is interpreter Clif
```

The simplest command is:

```
export_type extern int cprintf();
cprintf(1,"%s\n","hello");
```

On the standard output is written:

```
hello
```

If you make a mistake in the statement, the interpreter announces it. The argument 1 means *stdout*.

2.2 Variables and arithmetic operations

The following program adds two numbers:

```
/* program for input and output of the text */

export_type extern int cprintf();
extern int cscanf();

int a,b,c;
cprintf(1,"%s\n","input value a");
cscanf(0,"%d",a);
cprintf(1,"%s\n","input value b");
cscanf(0,"%d",b);
c=a+b;
cprintf(1,"the sum is %d\n",c);
exit;
```

The comment lines are embedded between the /* and */. They are unimportant during the run-time.

Variables have to be declared before they are used first time. If you forget the declaration statement the interpreter announces the mistake. The declaration contents of a name of the type and list of variables separated by commas:

```
int a,b,c;
```

The argument 0 in cscanf statement means *stdin*. The rest of the statements is described in the following sections.

Chapter 3

Data types, operators and expressions

3.1 Names of the variables

Names of the variables are sequences of characters. The first character must be a letter. The character "_" is assumed like a letter as well. The names of the variables are restricted. The names can not be keywords (e.g. if, else, int). The keywords are dedicated.

3.2 Data types

The interpreter Clif as well as the C-language, has only few fundamental data types:

char	one byte, it is possible to store one character from the set
int	integer, it is depending on a size of the integer of the host
float	single precision real number
double	double precision real number

3.3 Constants

There are different types of constants. The integer constant has usual notation for example 134 e.g. it is any sequence of digits. The float and double constants have the following notations: 134.435 or 12.56e-3 or 0.2345E3

The character constant is a character cited in single quotas, for example 'a'. The value of the character constant is integer value from the set of characters of the host computer (for example ASCII).

There are special character constant comprised of sequence of characters, for example: \n (newline), \t (tab), \\ (backslash), etc.

The string constant is embedded by double quotas, e.g. "I am a string". The quotas are not included in string constants. The string is an array of characters. At the end of the string constant an empty character \0 is appended. Therefore it is necessary to bear into mind that 'x' is different from "x".

3.4 Declaration

All variables must have been declared before they are used. The declaration contents of type specifier followed by a list of one or more variables of the type. For example:

```
int a,b,c;
double d,e,f;
int g;
double h;
```

The declaration is a little bit larger but it can be completed by a comment.

3.5 Arithmetic operators

The arithmetic operators are $+, -, *, /$ and the modulus operator %. There is a operator of unary $-$ but not unary $+$. The rest is truncated in the integer division. The expression $x\%y$ is the operation x divided by y. The result is zero, if y is the divisor of the x.

The operators $+$ and $-$ have the same precedence. The precedence is lower then the (equal) precedence of the operators $*, /$ and %. The arithmetic operators are left associative.

3.6 Relational and logical operators

The relational operators are
$> \quad >= \quad < \quad <=$
Each of them has the same precedence. The operators of equality are behind them in the precedence
$== \quad != $
(both have the same precedence). The relational operators have the lower precedence than the arithmetic ones.

The logical operators && and || are left associative. The logical expressions are evaluated from left to right.

The precedence of the operator && is higher than the operator ||. Both of them have the lower precedence than the relational and equality operators.

3.7 Type conversion

In an expression, the operands can differ in types. The coercions take place in the expressions which have a sense. For example, using a variable which has type double as an array subscript has no sense; therefore the coercion is not processed. In this case the interpreter announces an error.

If one of the operands is a double, all operands become double. Always, types of both processed operands are compared. Then the coercion is made to the type of operand which is "wider". The order of the implemented types from the "narrowest" to the "widest" is as follows: *char, int, float, double.* Another type of coercion is in an assignment. The right side value of the expression is converted to the type of the variable on the left side. The type of result is the same as type of the variable on the left side of the expression.

The type of the result can be forced by cast operator which causes an explicit conversion. In a statement
(name_of_type) expression
the *expression* is converted to the *name_of_type.*

3.8 Increment and decrement operators

The language of the interpreter has operators for the increment and decrement of the variables. The increment operator (++) adds one to the operand. The decrement operator (−−) subtracts one from the operand. The operators can be used only in the prefix notation. E.g. let n is 5, the expression

```
x = ++n;
```

assigns to the variable n the value 6. Then the value 6 is assigned to the variable x.

3.9 Bitwise logical operators

The Clif has operators for bit manipulating

&	the bitwise AND operator
|	the bitwise inclusive OR operator
^	the bitwise exclusive OR operator
<<	the left shift operator
>>	the right shift operator
~	the unary one's complement operator

The bitwise AND operator (&) is often used as a mask of the bits. The bitwise inclusive OR operator (|) is used in a setting of bits.

An user should bear in mind the big difference between the bitwise logical operators &, | and logical operators &&, ||. For example if x is 1 and y is 2 then expression $x\&y$ is equal zero, while $x\&\&y$ is equal one. The result of bitwise logical AND and bitwise logical OR is always *int*.

The shift operators $<<$ and $>>$ cause the shift of the left operand to the left or right. The number of shifted bits is in the right operand. The result of these operations is undefined if the right operand is negative or greater than or equal to the length of object in this bits.

The unary one's complement operator ˜ changes each bit, ones to zeros and vice versa. The type of the operand must be integer.

3.10 Conditional expression

There is a short form for the conditional statement using only expressions instead of statements, i.e.:

expr1 ? expr2 : expr3;

The expr1 is evaluated. If it is true, the expr2 is executed. If the expr1 is false, expr3 is executed.

3.11 Assignment operators

There are the following compound assign operators:

$*=$
$/=$
$\%=$
$+=$
$-=$
$<<=$
$>>=$
$\&=$
$\hat{}=$
$\vert=$

3.12 The precedence and the order of the evaluation

The following table offers the survey of the operators. The operators in the same line have the same precedence. The precedence increases line by line.

Operator	**Associativity**
$= \; += \; -= \; *= \; /= \; \%= \; \&= \; \hat{}= \; \vert= \; <<= \; >>=$	right to left
$?\,:$	right to left
$\vert\vert$	left to right
$\&\&$	left to right
$\vert$	left to right
$\hat{}$	left to right
$\&$	left to right
$== \; !=$	left to right
$< \; <= \; > \; >=$	left to right
$<< \; >>$	left to right
$+ \; -$	left to right
$* \; / \; \%$	left to right
$\tilde{}\,! \; ++ \; -- \; - \; (name\_of\_type)$	right to left
$(\,) \; [\,]$	left to right

The table must be born in mind to write correct program without complications.

Chapter 4

Control flow

4.1 Conditional statement

The conditional statement if-else has the following syntax:

```
if ( expr )
        { stat1 }
else
        { stat2 }
```

The else part is not compulsory. The expr is evaluated and if it is nonzero, the first substatement is executed. If the expr is equal zero and if the else part exists, the second substatement is executed (stat2).

4.2 Statement switch-case

If more conditional statements have to be used in a sequence, a switch can be used. Then the different branches can be followed using case statement. I.e.,

```
switch ( expr )
            {
            case expr1 : stat1
            case expr2 : stat2
            default : stat3
            }
```

The expr is evaluated. Then it is compared to expr1. If they are equal then the stat1 is executed. If they are not equal the execution continues at the next label (expr2 in this case). The expressions in labels have to be constant expressions. default is executed only if no expression in label statements is equal to the expr. The default statement can be omitted. A usage is in the following example:

```
export_type extern int cprintf();
extern int cscanf();
int a;
cprintf(1,"%s\n","input a");
cscanf(0,"%d",a);
switch (a)
{
default:
        cprintf(1,"%s\n","default");
case 1:
        cprintf(1,"%s\n","1");
        break;
case 2|5:
        cprintf(1,"%s\n","2|5");
        break;
case 3:
        cprintf(1,"%s\n","3");
```

```
        cprintf(1,"%s\n","no break");
case 4:
        cprintf(1,"%s\n","4");
        break;
}
exit;
```

4.3 while, do while and for loops

In

```
while ( expr )
            { stat }
```

the expr is evaluated. If its value is nonzero, the stat is executed and the expr is evaluated again. The loop continues until the expr becomes 0. After loop finishing, it is proceeded below the stat.

The sequence of statements

```
do
   { stat }
   while ( expr )
```

is executed at least once. The while expr is evaluated at the end of the loop sequence. If the expr is true the sequence of statements in the loop is repeated.

The statement for

```
for ( expr1; expr2; expr3)
                          { stat }
```

is equal with the sequence of the following commands

```
expr1;
while ( expr2 )
              { stat
              expr3; }
```

All three parts of the statement for are expressions. The expressions expr1 and expr3 are usually assignments. The expr2 is a relational expression. Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

If all three parts are dropped, the expr2 is considered true and

```
for ( ; ; ) {
           ...
           }
```

is an infinite loop. It is assumed another termination of the loop. For example by the statements break or return.

Everyone should consider which loop statement to use. If we need not to initialize or to reinitialize the loop variable, we can take an advantage of the statement while.

The loop statement for is useful if we need to initialize and reinitialize the loop variable. The control statements are concentrated at the beginning of the loop.

4.4 Statement break

It is useful to break processing of the loop at arbitrary place. The statement break causes termination of the most inner while or for. The following program shows function of the break:

```
export_type extern int cprintf();

int x;
x=1;
while (x<=10)
        {
        cprintf(1,"%s\n","now in while");
                if(x == 5)
                        {
                        cprintf(1,"%s\n","now in IF");
                        cprintf(1,"%s\n","*********************************");
                        x++;
                        break;
                        }
                cprintf(1,"x=%d\n",x);
                x++;
        }
exit;
```

Control passes to the statement following the terminated compound statement.

4.5 Statement continue

Statement continue is related to the statement break. It causes control to pass to the loop-continuation part of the most inner loop (for, while), i.e. to the end of the loop. The while loop proceeds at the expression. The for loop proceeds at the reinitialization. The program with the statement continue follows:

```
export_type extern int cprintf();

int x;
for (x=1;x<=10;x++)
        {
        cprintf(1,"%s\n","now in for");
                if(x == 5)
                        {
                        cprintf(1,"%s\n","now in IF");
                        print(1,"%s\n","*********************************");
                        continue;
                        }
                cprintf(1,"x=%d\n",x);
        }
exit;
```

and more complex example:

```
/* test program for nested loop */
export_type extern int cprintf();

int x,y;
y=1;
for(x=1;x<15;x++)
        {
         cprintf(1,"%s\n","now in for");
         cprintf(1,"x=%d\n",x);
         while(x>5 && x<10)
                {
                 cprintf(1,"%s\n","now in while");
                 cprintf(1,"x=%d\n",x);
                 if(x==9)
                        {
                         cprintf(1,"%s\n","now in the body of the first condition");
                         cprintf(1,"x=%d\n",x);
                         continue;
                        }
                 if(x==8)
```

```
                         {
                          cprintf(1,"%s\n","now in the body of the second condition");
                          break;
                         }
                  x=x+y;
                  cprintf(1,"x=%d\n",x);
                 }
         if(x==14)
                 {
                  cprintf(1,"%s\n","now in the body of the third condition");
                  cprintf(1,"x=%d\n",x);
                  break;
                 }
         }
 exit;
```

4.6 Statement goto

Although the goto statement is a part of unstructured programming the Clifenvironment supports this syntactic construction as well. The goto statement is used for unconditional branching of programs. The jump is possible only in the level one. There is no possibility to jump with goto statement across functions. Typical usage is in the following example:

```
 export_type extern int cprintf();

int i;

cprintf(1,"%s\n","in front");
  {
   a: cprintf(1,"%s ","a");

   if (10 == i)
     {
      cprintf(1,"%s ","in if"); ++i;
      goto b;
     }
   else
     {
      ++i;
     }
   goto a;
b:
  cprintf(1,"%s ","b");
  }
cprintf(1,"%s\n","after");
exit;
```

Chapter 5

Functions and program structure

Bigger tasks can be split by functions to the parts. Programmer can use the parts which were designed by other authors.

5.1 Fundamentals

Each function looks like

type name (type argument, ...)
{
declarations a statements
}

(The types are mentioned on the page 2. In addition, the type void can be in a function declaration). The void type denotes an nonexistent return value. Some parts of functions can be dropped out; minimal function is
type dummy() { }
. (The function can be useful in filling place in program design).

return statement returns value of the called function to the calling function. Syntax of the statement return is as follows:
return (expr);

The calling function can ignore the return value.

5.2 Scope rules

Global variables must have been declared before they are used. Functions must be declared before the formal call as well. The functions must be defined before the real call. The part of program where the name is declared is a scope of the name. The global variables are valid during whole interpreter session. The local variables are only valid during the processing of the function. The interpreter searches the table of local variables for the variable. If it is not successful, it proceeds searching the table of global variables for the variable. If it is not successful again it announces an error.

5.3 Recursion

Functions can be written in recursive form (e.g. a function can call *itself*). The recursion can be direct or indirect. An example of the recursion is n! (factorial):

```
export_type extern int cprintf();
extern int cscanf();

int n;
int fak(int n)
{
if(n==1)
```

```
        {return(1);}
return(fak(n-1)*n);
}
cprintf(1,"%s\n","input n");
cscanf(0,"%d",n);
n=fak(n);
cprintf(1,"factorial =%d\n",n);
exit;
```

Other examples follow:
Recursive function for Fibonacci sequence

```
export_type extern int cprintf();
extern int cscanf();

int n;
int fib(int n)
{
if(n<=1)
        {
        return(1);
        }
else
        {
        return(fib(n-1)+fib(n-2));
        }
}
cprintf(1,"%s\n","input n");
cscanf(0,"%d",n);
n=fib(n);
cprintf(1,"n=%d\n",n);
exit;
```

or for $\binom{n}{k}$

```
export_type extern int cprintf();
extern int cscanf();

int n,k;
int komb(int n,int k)
{
if((n==k) || (k==0))
        {
        return(1);
        }
else
        {
        return(komb(n-1,k)+komb(n-1,k-1));
        }
}
cprintf(1,"%s\n","input n");
cscanf(0,"%d",n);
cprintf(1,"%s\n","input k");
cscanf(0,"%d",k);
n=komb(n,k);
cprintf(1,"n=%d\n",n);
exit;
```

and for $\binom{n}{k} \cdot \sum_{i=1}^{l} i$

```
export_type extern int cprintf();
extern int cscanf();

int n,k,l;
int f(int n,int k)
{
if((k==0) || (n==k))
        {
        return(1);
        }
return(f(n-1,k)+f(n-1,k-1));
}
int komb(int n,int k,int l)
```

```
{
if((k==0) || (n==k))
        {
        return(1);
        }
if(l==1)
        {
        return(f(n,k));
        }
return(komb(n-1,k,l-1)+f(n-1,k)*l+komb(n-1,k-1,l-1)+f(n-1,k-1)*l);
}
cprintf(1,"%s\n","input n");
cscanf(0,"%d",n);
cprintf(1,"%s\n","input k");
cscanf(0,"%d",k);
cprintf(1,"%s\n","input l");
cscanf(0,"%d",l);
n=komb(n,k,l);
cprintf(1,"n=%d\n",n);
exit;
```

5.4 Including files for interpretation during the session

Users can include a file into the interpreter by statement
load (name_of_file);
The file will be interpreted automatically.

Chapter 6

Derived variable types

6.1 Arrays

One and multidimensional type of arrays are available in the interpreter. They are used to group the sets of the like variables. Example of one-dimensional array follows:

```
type name_of_array[number_of_elements];
```

The size of an array must be an integer constant. An element of the array can be accessed through subscripting. For example:

```
a[7]=56;
```

In the interpreter, subscripting begins with zero. An integer expression can be between the brackets; i.e. a subscripting variable, an expression, or return value from a function call.

6.1.1 Internal representation of arrays

Arrays are containing certain data type. They are stored in a contiguous set of these data types. See figure 6.1

6.2 Multidimensional arrays

In the interpreter, arrays of arbitrary dimensions can also be declared. The two-dimensional array with $m \times n$ elements is declared as follows:

```
type name_of_array[m][n];
```

An element of the array can again be accessed through subscripting, e.g.

```
a[7][9]
```

This will access the element in the 8-th row (row number 7) and in the 10-th column (column number 9).

6.3 Array names in expressions

The element of the array in an expression can only be accessed through subscripting, but there is an exception; the array can be passed as a parameter to a function (without subscripting). An example of the parameter passing mechanism using an array as a parameter follows:

```
export_type extern int cprintf();

int a[5][5][5][5][5];
a[0][0][0][0][0]=1;
a[1][1][1][1][1]=1;
a[2][2][2][2][2]=1;
a[3][3][3][3][3]=1;
```

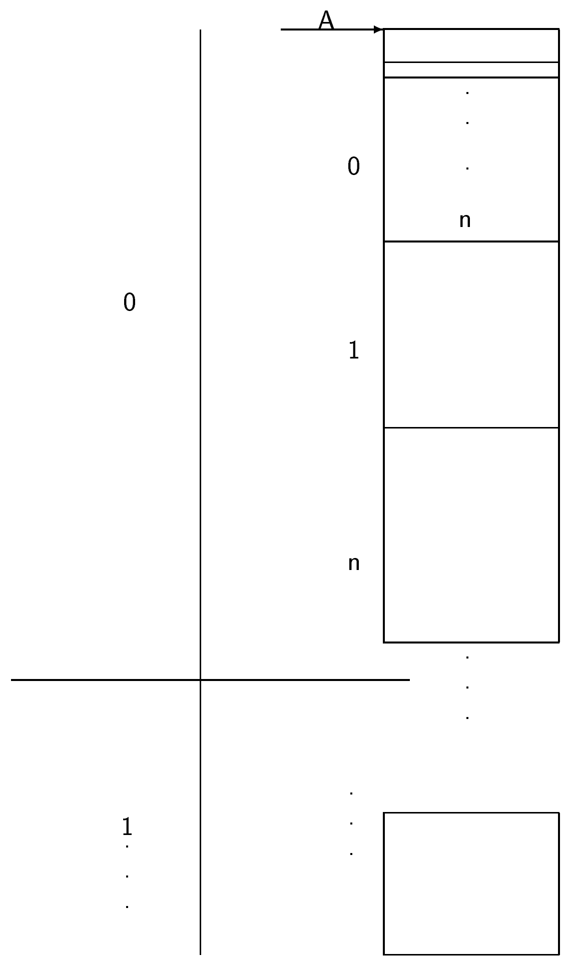

Figure 6.1: Internal representation of arrays

```
a[4][4][4][4][4]=a[3][3][3][3][3];
int z(int b[][5][5][5][5])
{
cprintf(1,"%d\n",b[0][0][0][0][0]);
cprintf(1,"%d\n",b[1][1][1][1][1]);
cprintf(1,"%d\n",b[2][2][2][2][2]);
cprintf(1,"%d\n",b[3][3][3][3][3]);
cprintf(1,"%d\n",b[4][4][4][4][4]);
}
z(a);
exit;
```

Chapter 7

Termination of the Clif session

Statement exit; terminates the interpreter session. This statement terminates correctly work of the interpreter environment.

Chapter 8

Run-string handling

So far, the input was always standard input. But the interpreter can have specified parameters in the run-string (command line) which are names of programs. The input is redirected and a file is opened instead of the standard input.

Other parameters can appear in the command line. One of the parameters is /help evoking brief help. Another parameter is for resetting of the size of main memory areas of the interpreter. The size is specified relatively. This parameter can be specified in the run-string as follows:

```
/bc=<number>
```

As the <number> can be specified a positive integer. The default value is 10.

Chapter 9

The standard I/O routines

So far, all our sample programs have written data to the standard output and read data from standard input (in both cases the terminal). As long as we are accessing not large portions of data, thus data in the simple form, this type of I/O routines is sufficient. However, large amounts of data are generally stored in files. To perform I/O from and to files I/O functions are included in the interpreter.

Access to files generally requires four basic functions:

open This enables access to a file and establishes the file handle.

close This terminates access to the file. When the access to a file is complete, it should be closed. The number of files that the interpreter can simultaneously manage is limited. Therefore the files should be properly closed.

read This function gets data from the file.

write This function adds information to the file or replaces information already in the file.

9.1 Opening a file

The standard I/O function used to open a file is named copen(). The function returns a handle to the opened file. The function has to be declared as follows:

```
extern int copen();
```

The function has two arguments:

- file_name is the starting address of a character string describing the name of the file. This can be a string constant or an array name.
- mode which describes the action that should be performing on the file. The modes are:
  - r The file is opened for reading started at the beginning of the file.
  - w The file is opened for writing; it is assumed that the file is to be created. If the file does not exist, it is created; if it exists, it is truncated (size of the file is reduced to zero) and positioned at the beginning of the file.
  - a The file is opened for writing. The mode is the same as mode for w except that the initial position is at the end of the file, i.e. the file is not truncated.
  - r+ The file is opened for update, i.e. the file is opened for reading and writing and is not truncated. The initial position is at the beginning of the file.
  - w+ The file is opened for update, i.e. the file is opened for reading and writing and is truncated, if the file already exists. If it does not exist, it is created. The initial position is at the beginning of the file.
  - a+ The file is opened for update, i.e. the file is opened for reading and writing and is not truncated and the initial position is at the end of the file.

The function returns an handle (integer number) if opening is successful. Otherwise the function returns EOF.

9.2 Closing a file

If the access to a file was completed, the file should be closed. The declaration of the function is as follows:

```
extern int cclose();
```

The argument of the cclose() function is the return value from the copen() function (the handle). The handle is released. If the program exits without closing a file, the system closes automatically the opened file.

The return values are 0 that means a successful file closing or *EOF* that means an error was encountered during the closing of the file.

9.3 Reading from a file

As far, we have only used the cscanf() function. Another function that can be used is cgetc() function. The parameter of the function is an integer number (handle), which was returned by a copen() function. The cgetc() function reads a character from the file which is pointed by the handle. The function cgetc() is declared as follows:

```
extern int cgetc();
```

This function returns an *EOF* character when an error occurred or at the end-of-file.

9.4 Writing to a file

In the previous sections we only write to the *stdout* via the cprintf() function. Analogous function to the cgetc() is a function cputc(). The function is declared:

```
extern int cputc();
```

The function writes a character into the file. On failure, the function returns an *EOF*.

9.5 Functions cscanf() and cprintf()

The functions used in previous section read from standard input and wrote to standard output. Default handles for *stdin*, *stdout* and *stderr* are 0,1,2 respectively. These functions can read from or write to a file. As the first parameter of the functions is used a handle of the file, for example:

```
float f;
int a,b;

extern int cscanf ();
export_type extern int cprintf ();

extern int cclose ();
extern int copen ();

cscanf(0,"%f",f);
b=copen("test_file","w");
cprintf(1,"handle = %d\n",b);
cprintf(b," f=%f\n",f);
cscanf(0,"%d",a);
cprintf(b,"integer value = %d\n",a);
cclose(b);
cprintf(1,"%s\n","I am a string");
exit;
```

The second parameter in the cscanf() function is a format string and the third parameter is a variable name for storing the input.

The second parameter of the cprintf() function is a format string and/or string constant and the third parameter is a variable name for output.

The following list describes valid format strings:

d the input/output field is a decimal integer; the corresponding variable name must point to the integer

u the input/output field is a decimal integer; the corresponding variable name must point to the integer; value is unsigned

o the input/output field is a octal integer; the corresponding variable name must point to the integer

x the input/output field is a hexadecimal integer; the corresponding variable name must point to the integer

e,f,g the input/output field is an optionally signed string of digits. The field may contain a radix character and an exponent field begins with a letter E or e, followed by an optional sign or space and an integer. The variable must point to the floating point variable. If you specify l, the variable must point to the double precision variable.

s the input/output field is a character string. The variable must point to an array of characters large enough to contain the string and a termination character (\0). The cscanf() function adds the termination character automatically. A white-space character terminates the input string, so the input string cannot contain spaces.

c the input/output field is a character or character string. The variable must point to either a character variable or a character array.

Chapter 10

Graphics interface

So far, the sample programs have written data to the standard output in the numerical form. The interpreter also provides graphics outputs. Certain type of graphics functions is included in the interpreter. A call to the graphics functions results in establishment of a channel. Access to channels generally requires four basic operations:

open First channel should be opened which allows access to the other graphics functions.

close This terminates access to the channel. When the access to the channel is complete, it should be closed. The number of channels that can simultaneously be opened is limited.

write This adds data to the channel.

flush This flushes all data from the channel to the output device.

10.1 Opening a channel

The graphics interface function used to open a channel is named chopen(). It returns a integer number - handle, if opening succeeds; otherwise it returns -1. The handle must be saved; all other graphics functions require the handle as an argument. The chopen() function should be declared as follows:

```
extern int chopen("arguments");
```

This function creates a window for a graphics output. This must be done before attempts are made to write in to the channel.

The chopen() has the following arguments (arguments are in the form of sublanguage):

fields This record is compulsive. This is an integer number which specifies number of records written to the channel. The other arguments have their default values.

style This specifies the type and path (attributes) of the drawn object. The style is a positive integer number.

lower This denotes lower boundary of the drawn object.

upper This denotes upper boundary of the drawn object.

print_format The available values are point and line. Default is line.

start_time Initial setup of the time. The available values are automatic and an arbitrary real number. The automatic means that the window is cleared when the output reaches the right end of the window. New initial time continues at the end value of the time in previous window. Default value is 0.0.

duration_time This means a length of the window, i.e. the number of the records contained in the window in the vertical direction. The available values are automatic and an arbitrary real number. The automatic means if the output reaches the right end of the window then output is redrawn, i. e. the duration_time size is changed. Default value is 1000.0.

w_resolution This specifies the size of the window. The default value is 256 by 256 pixels.

type This specifies if the alphanumeric window should be created as well. The available values are alpha and graphic. Default value is the graphic.

OnLeaveW The parameter specifies if the interpreter should be stopped temporarily when enriches the right end of the window. The available values are noevent and suspend. The default value is the noevent.

The records are separated by a backslash immediately followed by a newline. The first record should be the fields. The other records can be arbitrarily ordered.

10.2 Closing a channel

When the program has completed its use of a channel, the channel should be closed using chclose() function. This function is declared as follows:

```
extern void chclose();
```

This function closes the specified channel and releases the handle. The handle is the only parameter of the chclose function. If program exits without closing a channel, the system will automatically close the channel.

10.3 Writing to a channel

The graphics interface function used for writing to a channel is named chwrite. This function is declared as follows:

```
extern void chwrite();
```

The function has the following arguments: The first argument is the handle of the channel in to which data should be written. The second argument is a datum. The data are collected in the specified channel. When number of records is equal to the number specified in the record fields (see page 21 for more details), the data are put into the window.

10.4 Flushing a channel

This function flushes a channel in which were collected data. The function is declared:

```
extern void chflush();
```

The function has no arguments. It flushes all opened graphics channels. This function is necessary at the end of putting image to the window, when data are in the channel but in the window.

The following program shows a use of graphics interface functions:

```
double x,y;

extern double sin();

/* declaration of graphics functions */
extern int chopen();
extern void chwrite();
extern void chclose();
extern void chflush();

int handle;

handle=chopen("fields=6\
lower(0)=-0.5\
upper (0) = 0.5\
style (0) = 10\
lower(1)=-1.5\
upper (1) = 1.5\
style (1) = 20\
lower(2)=-3.\
upper (2) = 3.\
style (2) = 15\
lower(3)=-3.5\
upper (3) = 3.5\
```

```
style (3) = 5\
lower(4)=-4.\
upper (4) = 4.\
style (4) = 17\
lower(5)=-4.5\
upper (5) = 4.5\
style (5) = 7\
start_time=automatic\
duration_time=256.\
w_resolution=600 400\
print_format=point\
type=alpha\
OnLeaveW=suspend");

for(x=0;x<6.5;x=x+0.01)
        {
        y=sin(x);
        chwrite(handle,y);
        chwrite(handle,y);
        chwrite(handle,y);
        chwrite(handle,y);
        chwrite(handle,y);
        chwrite(handle,y);
        }
chclose(handle);
exit;
```

Chapter 11

Interrupt handling

The run of the virtual machine of the interpreter can be interrupted. The interpreter provides two kinds of the interrupts; synchronous and asynchronous.

11.1 Synchronous interrupt

The synchronous interrupt is specified in the form of statements in a program. The following statement can be placed where it is useful:

```
csuspend;
```

When the interpreter reaches the statement it is stopped and echoed:

```
clif interrupt level <number>
```

Where the <number> is a positive number. A user, at any interrupt level, can do anything what is available in the interpreter. To resume the run of the interpreter a user should write the following statement:

```
resume;
```

or press the key DC4 (Ctrl-t). The interpreter is then echoed:

```
clif interrupt level <number>-1
```

which means that the interrupt level is decreased. The lowest interrupt level is zero. An example of use of the synchronous interrupt follows:

```
int a,n;
extern int cscanf();
export_type extern int cprintf();
int fakt(int n)
{
csuspend;
if(n==1)
        {
        return(1);
        }
else
        {
        return(fakt(n-1)*n);
        }
}
cprintf(1,"input n %d\n",n);
cscanf(0,"%d",n);
csuspend;
a=fakt(n);
cprintf(1,"factorial = %d\n",a);
exit;
```

11.2 Asynchronous interrupt

An asynchronous interrupt can be invoked by the user with pressing the key Ctrl-t while the virtual machine of the interpreter is running. A dummy program in which a user can follow the use of asynchronous interrupt is:

```
extern int cscanf();
export_type extern int cprintf();

int i;
for(i=0;i<100000;++i)
        {
        cprintf(1,"%d",i);
        }
cprintf(1,"%s\n","loop is over");
exit;
```

After making statements requested by the user, the further processing can be invoked as by the synchronous interrupt with pressing Ctrl-t or typing resume.

Chapter 12

Errors

12.1 Clif error messages

In this chapter is a list of Clif error messages.

12.1.1 Syntax error messages

```
    case 1000:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"variable ‘%s’ isn’t declared\n",
text);
      break;
    case 1001:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"variable ‘%s’ was already declared\n",
text);
      break;
    case 1002:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"remote procedure %s is not declared\n",
proc_name_text[proc]);
      break;
    case 1003:
      ERROR_INFO;
      fprintfx (stderr,
"local variable ‘%s’ was already declared\n",
text);
      break;
    case 1004:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"at the %d-th char, near the ‘%s’\n",
char_counter, yytext);
      break;
    case 1005:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid type of the operand, %d-th character\n",
char_counter);
      break;
    case 1006:
      ERROR_INFO;
      fprintfx (stderr,
"remote function ‘%s’ already declared\n",
text);
      break;
    case 1007:
      ERROR_FULL_INFO(line_counter);
```

```
      fprintfx (stderr,
"remote function isn't declared\n");
      break;
    case 1008:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"remote functions are not in the load table\n");
      break;
    case 1009:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"`void' type in expression\n");
      break;
    case 1010:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"`void' type assigned to l_value\n");
      break;
    case 1011:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"load can't open file `%s'\n",
yytext);
      break;
    case 1012:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"variable or field `%s' declared void\n",
text);
      break;
    case 1013:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"switch quantity not an integer\n");
      break;
    case 1014:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"case label does not reduce to an integer constant\n");
      break;
    case 1015:
      ERROR_FULL_INFO(tmp_c->line_number);
      fprintfx (stderr,
"duplicate case value\n");
      ERROR_FULL_INFO(tmp_m->line_number);
      fprintfx (stderr,
"this is the first entry for that value\n");
      break;
    case 1016:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"case label not within a switch statement\n");
      break;
    case 1017:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"struct tag `%s' was already declared\n",
text);
      break;
    case 1018:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"union tag `%s' was already declared\n",
text);
      break;
    case 1019:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"enum tag `%s' was already declared\n",
text);
      break;
    case 1020:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"conversion to non-scalar type requested\n");
```

```
      break;
    case 1021:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid type argument of ‘->’\n");
      break;
    case 1022:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid lvalue in unary ‘&’\n");
      break;
    case 1023:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"storage size of ‘%s’ isn’t known\n",
text);
      break;
    case 1024:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"parameter ‘%s’ has incomplete type\n",
text);
      break;
```

12.1.2 Clif compilation error messages

```
    case 2000:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid number of subscripts\n");
      break;
    case 2001:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"‘%s’ is not an array variable\n",
text);
      break;
    case 2002:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid type of array subscript\n");
      break;
    case 2003:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"type of formal parameter does not match previous declaration\n");
      break;
    case 2004:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"number of formal parameters does not match previous declaration\n");
      break;
    case 2005:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"name of formal paramter does not match previous declaration\n");
      break;
    case 2006:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"size of array subscript of formal parameter does not match previous declaration\n");
      break;
    case 2007:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
```

```
"number of array subscripts of formal parameter does not match previous declaration\n");
      break;
    case 2008 :
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"structure has no member named ‘%s’\n",
text);
      break;
    case 2009:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"request for member ‘%s’ in something not a structure or union\n",
text);
      break;
```

12.1.3 Clif

```
    case 3000:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"‘break’ outside loop or switch\n");
      break;
    case 3001:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"bad used ‘continue’\n");
      break;
    case 3002:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"default label not within a switch statement\n");
      break;
    case 3003:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"multiple default labels in one switch\n");
      ERROR_FULL_INFO(fixp->switch1.def_use.line_number);
      fprintfx (stderr,
"this is the first default label\n");
      break;
    case 3004:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"duplicate label ‘%s’\n",
text);
      break;
    case 3005:
      ERROR_FULL_INFO(error_line_number);
      fprintfx (stderr,
"label ‘%s’ used but not defined\n",
text);
      break;
    case 3006:
      ERROR_FULL_INFO(error_line_number);
      fprintfx (stderr,
"invalid lvalue in assignment\n");
      break;
```

12.1.4 Clif run-time error messages

```
    case 4000:
      ERR_NO_INFO;
      fprintfx (stderr,
"interpreter: full memory\n");
      break;
    case 4001:
      ERR_NO_INFO;
```

```
      fprintfx (stderr,
"interpreter: stack overflow\n");
      break;
    case 4002:
      ERR_NO_INFO;
      fprintfx (stderr,
"operating system out of memory\n");
      break;
```

12.1.5 Clif fatal error messages

```
    case 5000:
      fprintfx (stderr,
"Compile Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
"Interpreter Internal Error (unknown operand type) in line %d e-mail: %s\n",
line_counter, EMAIL);
      print_source_line ();
      break;
    case 5001:
      fprintfx (stderr,
"Run-time Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
" Internal Interpreter Error (unknown instruction) e-mail: %s\n",
EMAIL);
      print_source_line ();
      break;
    case 5002:
      fprintfx (stderr,
"Compile Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
" Interpreter Internal Error (error in book-keeping) in line %d e-mail: %s\n",
line_counter, EMAIL);
      print_source_line ();
      break;
    case 5003:
      fprintfx (stderr,
"Compile Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
" Internal Interpreter Error (error in operand type) in line %d e-mail: %s\n",
line_counter, EMAIL);
      print_source_line ();
      break;
```

12.1.6 Clif warning messages

```
    case 6000:
      if (warning_yes)
{
  ERROR_FULL_INFO(line_counter);
  fprintfx (stderr,
    "remote function %s already declared\n",
    text);
}
      return;
    case 6001:
      if (warning_yes)
{
  ERROR_FULL_INFO(line_counter);
  fprintfx (stderr,
    "`return' with no value, in function returning non-void\n");
}
```

```
      return;
    case 6002:
      if (warning_yes)
{
  ERROR_FULL_INFO(line_counter);
  fprintfx (stderr,
    "‘return’ with a value, in function returning void\n");
}
      return;
    case 6003:
      if (warning_yes)
{
  if (proc)
    {
      print_file_name ();
      fprintfx (stderr,
" In function ‘%s’:",
proc_name_text[proc]);
      ERROR_INFO;
      fprintfx (stderr,
"unused variable ‘%s’\n",
text);
    }
  else
    {
      print_file_name ();
      fprintfx (stderr,
" In block finishing at line %d:\n",
line_counter);
      ERROR_INFO;
      fprintfx (stderr,
"unused variable ‘%s’\n",
text);
    }
}
      return;
    case 6004:
      if (warning_yes)
{
  ERROR_FULL_INFO(error_line_number);
  fprintfx (stderr,
    "label ‘%s’ defined but not used\n",
    text);
}
      return;
    case 6005:
      if (warning_yes)
{
  ERROR_INFO;
  fprintfx (stderr,
    " ‘/*’ within comment\n");
}
      return;
    case 6006:
      if (warning_yes)
{
  if (proc)
    {
      print_file_name ();
      fprintfx (stderr,
" In function ‘%s’:",
proc_name_text[proc]);
      ERROR_INFO;
      fprintfx (stderr,
"‘%s’ might be used uninitialized in this function\n",
text);
    }
  else
    {
      print_file_name ();
      fprintfx (stderr,
" In block finishing at line %d:",
line_counter);
      ERROR_INFO;
```

```
      fprintfx (stderr,
"`%s' might be used uninitialized in the block\n",
text);
    }
}
      return;
    case 6007:
      if (warning_yes)
{
  print_file_name ();
  fprintfx (stderr,
    " In function `%s':",
    proc_name_text[proc]);
  ERROR_INFO;
  fprintfx (stderr,
    "number of locals is greater than the ANSI allows\n");
}
      return;
    case 6008:
      if (warning_yes)
{
  print_file_name ();
  fprintfx (stderr,
    " In function `%s':\n",
    proc_name_text[proc]);
  ERROR_INFO;
  fprintfx (stderr,
    "number of params is greater than the ANSI allows\n");
}
      return;
    case 6009:
      if (warning_yes)
{
  char *tmp_line, *beg, *end, *com;
  int n;
  n = strlen(line_buf);
  tmp_line = malloc(n+1);
  if (NULL == tmp_line)
    {
      perror ("");
      abort ();
    }

  strcpy (tmp_line, line_buf);
  beg = strrchr (tmp_line, '(');
  if (NULL == beg)
    {
      perror ("");
      abort ();
    }

  beg++;
  com = strrchr (beg, ',');
  if (NULL != com)
    beg = com + 1;
  for (; *beg == ' ' || *beg == '\t'; beg++);

  end = strrchr (beg, ' ');
  com = strchr (beg, ' ');
  if (end != com)
    *end = '\0';
  else
    tmp_line[n - 1] = '\0';

  ERROR_INFO;
  fprintfx (stderr,
    "`%s' declared inside parameter list its scope is only this definition or declaration, which is probably not what you want.\n",
    beg);
  free (tmp_line);
}
      return;
```

12.1.7 Clif initialization error messages

```
    case 7000:
      print_error_number (err_no);
      fprintfx (stderr,
"in run-string and/or in ‘clif.ini’ file\n");
      break;
    case 7001:
      print_error_number (err_no);
      fprintfx (stderr,
"interpreter: can’t open file %s\n",
argvv[argc_counter]);
      break;
    default:
      fprintfx (stderr, "Fatal error invalid error number (%d) e-mail: %s\n", err_no, EMAIL);
      break;
```

Bibliography

[1] A. V. Aho and S. C. Johnson. LR parsing. *Computing Surveys*, 6(2), June 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1, 2. Prentice Hall, Englewood Cliffs, 1972, 1973.

[4] S. Bandyopadhyay, V.S. Begwani, and R.B. Murray. Compiling for the crisp processor. In *Proceedings of the IEEE Spring, COMPCON*, pages 96–100, 1987.

[5] A.D. Berenbaum, D.R. Ditzel, and H.R. McLellan. Introduction to the crisp instruction set architecture. In *Proceedings of the IEEE Spring, COMPCON*, pages 86–90, 1986.

[6] A.D. Berenbaum, D.R. Ditzel, and H.R. McLellan. Architectural innovations in the crisp microprocessor. In *Proceedings of the IEEE Spring, COMPCON*, pages 91–95, 1987.

[7] J.J. Dongarra, C.B. Moler, and et al. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.

[8] D. Gries. *Kompilátory číslicových počítačov.* Alfa, 1981.

[9] D.G. Kafura and R.G. Lavender. Concurrent object-oriented language and the inheritance anomaly. In *Proceedings ISIPCALA'93*, pages 183–215, Prague, 1993.

[10] B. W. Kernighan and D. M. Ritchie. *Programovací jazyk C.* Alfa, 1988.

[11] B. Serpette, J. Vuillemin, and J.C. Hervé. BigNum: A portable and efficient package for arbitrary-precision arithmetic. Research report, INRIA, 1989. preprint.

[12] Ľ. Koreň. Small interpreter on the basis of C-language. Technical report, Slovak Technical University, Faculty of Mechanical Engineering, Bratislava, 1993.

[13] N. Wirth. *Algoritmy a štruktúry údajov.* Alfa, 1989.

Index