RedCrab^{PLUS}

The Calculator

Programmers Manual

Version 4.20

Content

1.1 Program Overview

RedCrab can open one or more source files (modules), each containing functions with series of statements or data. This manual describes the syntax rules and gives an overview of data access per text files and the program editor.

1.2 Programming

The *RedCrab* Calculator has its own simple program language implemented. It is easy to learn, so users without programming experience can write their own functions in a short time.

According to the worksheet, *RedCrab* makes no distinction by keywords and system function, between uppercase and lowercase letters. Names of own defined functions and variables are case-sensitive.

The program code ignores any extra spaces, tabs, linefeed and comments. When necessary, it uses the keyword *end* in association with the statement type to terminate a statement. Apart from this, it has no statement terminators such as semicolon. The end-of-line terminates a statement. For exceptions read the description about multi rows statements below.

Example: **Let** a = 12
             Let b = 22

You can write several statements in one line if they are separated by a colon.

Example: **Let** a = 123 : **Let** b = 22

1.3 Comments

For clarity and to simplify programmes, it is recommended that you document your codes by including comments. You can also use comment symbols during program development to disable statements without deleting them. You can indicate comments in two ways:

- A comment can begin with the left parenthesis symbol following with the multiplication symbol **(*** and end with the symbols *****)**. You cannot use the symbols to nest comments.

- You can also use a double slash symbol **//** . The comment terminates at the end of the line.

1.4 Identifiers

RedCrab programs can reference modules, functions, local and global variables and constants. With the exception of constants, each of these must have an identifier as a name. An identifier is a sequence of letters, digits and underscores. The first symbol must be a letter or underscore.

1.5 Scope of Identifier

Local variables must be defined within a function. They cannot be referenced by statements outside their function.

Global variables must be defined outside a function. They can be referenced by all statements in the defined module. Other modules and worksheet can read them.

Functions can be referenced by all statements in modules and worksheet.

1.6 String Constants

String constants are sequences of character enclosed within double quotes. The constant must be written in a single line. You can create longer strings by concatenating string constants with the ***Point*** (.) symbol.

Example: **Let** s = "Hallo "
            Let t = s ."Welt"

The variable t above contains: "*Hello World*"

1.7 Program Variables

You must define a program variable before you can use this variable. You do this by assigning an identifier to the keyword *define*. The *define* statement allows optional assignment of a value. For more information, read the description about *define* below.

1.8 Number Expressions

A number expression consists of a number constant, variable, cells of a field, function that returns a number value or several of these, connected by one of the following arithmetic operators:

```
*     Multiplication
/     Division
Mod   Modulo
Div   Integer Division
+     Addition
-     Subtraction
```

1.9 Boolean Expressions

A Boolean expression evaluates either *TRUE* or *FALSE* and has the following form:

expression operator expression

The expressions can be numeric or text strings. If a numeric expression is compared with a string containing a number, the value of the number is considered. If two strings are compared, they are always treated as strings, regardless of whether they contain text or numbers. Operator is one of the following relational operators:

Operator	Operation
==	Equal to
< >	Not equal to
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

Boolean expression can be compounded with the operators *AND(&)* and *OR(|)*.

Examples:
```
(a >= b) & (c <= d)
(a == d) |  c
```

1.10 Expressions

For *if* and *while* statements, *TRUE* is any non-zero number and *FALSE* is zero. In these statements, you can use a number expression where a Boolean expression is called for. You can use a Boolean expression where a number expression would be expected, yielding 1 or 0. You can use a string expression that is a representation of a number anywhere that a number expression is allowed.

1.11 Multiple Line Statements

A field definition can be continued on the next line. The current line must end with a comma or a semicolon.

A statement may be continued on to the next line if current line ends with backslash.

A long number can be continued on to the next line if current line ends with a double backslash "\ \".

The following table shows some examples of multi-line statement:

| Statements	Interpretation
`Let m = [1,2,3;` `        4,5,6]`	`Let m = [1,2,3;4,5,6]`
`Let m = [1,2,3,4,` `        5,6,7,8]`	`Let m = [1,2,3,4,5,6,7,8]`
`Let v \` `  = 1 + 2`	`Let v = 1 + 2`
`Let v = 12345\\` `        6789`	`Let v = 123456789`
`Let s = "hello " \` `    + "world"`	`Let s = "hello " + "world"`

2.1 Program

A program file always starts with the key word *program*. Optionally, a program name can be specified. The program file is displayed in a tab with their file name, or a specified program name.
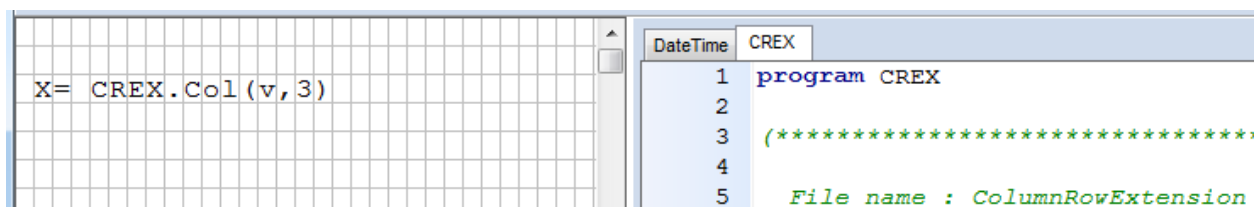
The syntax of the *program* statement is:

```
Program
Program name
```

The following example shows how to call the function Col in module *ColumnRowExtension* from the worksheet. The program file is qualified with the name of the register.
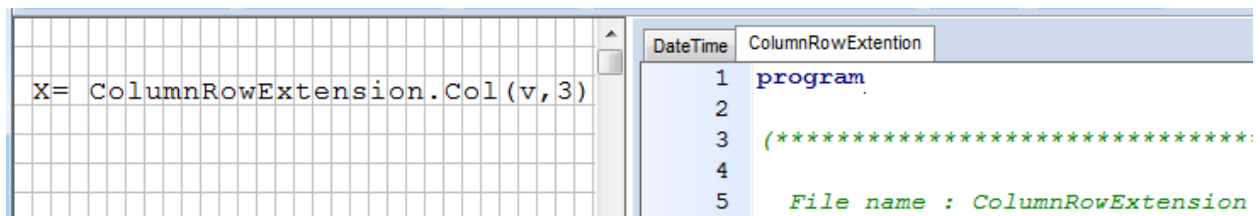
More comfortable is the call of the function *Col* in the following example, where a short program name is specified. When loading the module from file, *RedCrab* named the tab like the program name instead the file name.

2.2 Uses

A *uses* clause lists modules used by the following program. The clause consists of the reserved word *uses*, followed by one or more comma delimited module names (file names). The uses statement must follow the program statement immediately. The syntax of the *uses* statement is:

```
uses module1, module2
```

2.3 Define

The *define* statement declares the name of a variable and assign a value. If you do not assign a value, the variable is initialized with Zero. You cannot reference a variable before it has been declared by the *define* statement, or in the functions parameter list.

The syntax of the *define* statement is:

```
define Name
define Name = Value
```

2.4 Let

Let assigns an expression to a variable. The syntax of the *Let* statement is:

```
let variable = Ausdruck
```

The expression consists of a constant, variable, cells of a field, function that returns a value or several of these. The value can be a simple number or Boolean value, a data field or a text string.

Example:

```
Let x = 12
Let x = y
Let x = (12 + y) * z
Let x = sin(45)
Let x = "hello"
```

2.5 While do

Use *While* when you want to repeat a set of statements. The syntax of a *while do* statement is:

```
while expression do
   statements....
end
```

While repeats the statements between *do* and *end* so long as the expression condition remains *True*. If it is *False*, program control passes to the statement following the *End* statement.

Example:

```
Let i = 0
While i < 100 do
   Statement Sequence....
   Let i = i + 1
End
```

2.6 If Then

The *if* controls conditional program branching. The statements between *then* and *end* is executed if the value of the expression is nonzero (*TRUE*). Otherwise, the statement block is skipped and the program continues with the statement following *end*.

The syntax of the *if* statement is:

```
if expression then
      statements....
end
```

Example:

```
if i < 100 then
  Let x = 10
End
```

2.7 Else

The *else* statement is an extension of *if then* statement. In the description of *if* statements above, the *statement* is ignored, if *expression* is null (*FALSE*). In this form of syntax, which uses the *else* statement, the statements between *else* and *end* are execute if *expression* is null (*FALSE*).

The syntax of *if then* e*lse* statement is:

```
if expression then
  statements....
else
  statements.
end
```

2.8 Elseif

With *elseif* statement, additional conditions for program branching can be programmed. The expression, following *elseif* is only evaluated if the preceding *if* and *elseif* expressions evaluate to zero (*FALSE*). If this *expression* has a nonzero value (*TRUE*) , then the following statements is executed until the next *elseif* or *else* statement. Then the program skips below the end statement.

The syntax of the *Elseif* statement is:

```
if expression then
   statements....
elseif
   statements.
elseif
   statements.
else
   statements.
end
```

2.9 Function

The *function* statement defines a function. A function is a named block of statements. The function can be invoked from the worksheet and any module of your program. The function returns a single value or a data field; it can be invoked as an operand within a *RedCrab* expression. Otherwise, you must invoke it with the *Call* statement.

The syntax of the *Function* statement is:

```
function Name  (argument, argument......)
   statements...
end
```

argument are the formal arguments to this function. You can reference the arguments without declaration by the *define* statement.

2.10 Forward

The purpose of a *forward* declaration is to extend the scope of a function identifier to an earlier point in the source code. This allows other functions to call the forward-declared routine before it is actually defined. Besides letting you organize

your code more flexibly, *forward* declarations are sometimes necessary for mutual recursions.

The syntax of the *Forward* statement is:

    forward *Name*

2.11 Call

The *call* statement invokes a specified function. No result is expected.

The syntax of the *call* statement is:

```
Call FunctionName (argument, argument...)
```

2.12 Result

Result return values to the calling routine. The return value can be a number, Boolean or string constant, variable, cells of a field, function that returns a value or an expression.

The syntax of the *Result* statement is:

```
Result = Value
```

3.1 PHP Interface

RedCrab include an integrated *PHP* developer environment. For instruction about the *PHP* installation and configuration read the *RedCrab*^{PLUS} manual. This chapter describe the *PHP* programmers interface for data transfer between *PHP* and *RedCrab*. The interface uses the *PHP* standard Input / Output.

3.2 PHP Input

RedCrab sends data to *PHP* programs with the method *POST* to the standard input. The data includes one or more named arguments. The first argument is named *func*, it contains the name of the function which is called from *RedCrab*. The next arguments are the function parameters, they are named as *arg1*, *arg2*, *arg3* and so on.

In the following example *RedCrab* calls the function *Add* in the *PHP* program *math* with two parameters.

Example:

Function call in *RedCrab*:

```
x = math.Add(a,b)
```

Handle the call in *PHP*:

```
$fname = $_POST["func"];        // get the function name
$a1    = $_POST["arg1"];        // get the first argument
$a2    = $_POST["arg2"];        // get the second argument
$fname($a1,$a2);                // call the function Add()
```

The example above calls a function with two numeric parameters. If you send arguments which contains data fields, *RedCrab* convert these into *PHP* format.

The next example shows different accesses to arrays in arguments. The *RedCrab* interface handles one- and two dimensional data fields.

Example:

Access to one-dimensional arrays

```
1.)   $data = $_POST["arg1"];
      $x    = $data[1];
      $y    = $data[2];


2.)   $x = $_POST["arg1"][1];
      $y = $_POST["arg1"][2];



      Access to multi-dimensional arrays

1.)   $data = $_POST["arg1"];
      $x    = $data[1][0];
      $y    = $data[2][0];

2.)   $x = $_POST["arg1"][0][1];
      $y = $_POST["arg1"][1][1];
```

3.3 PHP Output

The PHP program sends data to *RedCrab* via *echo* statement. The following example shows the handling of a numeric value.

Example: `echo $val;`

If a function returns a data field (*PHP* array), the output string must be formatted equivalent to the *RedCrab* data file format. This means the interface transfers the data field row for row. The columns are delimited with commas; the semicolon delimits the rows.

The *PHP* file *RCFieldEcho.php* contains a function which handles the output formatting. See the code below. You can copy or include this file in your program.

```php
<?PHP

function RCFieldEcho($a)
{
  $cnt = count($a);
  $mcnt = count($a[0]);
  for ($i = 0; $i < $cnt; $i++)
  {
    if ($mcnt > 1 )
    {
      if ($i > 0) echo ';';
      RCFieldEcho($a[$i]);
    }
    else
    {
      if($i > 0) echo',';
      echo $a[$i];
    }
  }
}

?>
```