RedCrab

Math V

Programmers Manual

Version 5.0

Content

1.1 Programming

This manual describes the syntax rules of the *RedCrab* program interpreter. *RedCrab* 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
```

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

Example:
```
/* comment */
// comment
```

1.3 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.

1.4 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.

1.5 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 = "Hello "
Let t = s ."World"
```

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

1.6 Program Variables

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

1.7 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

1.8 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
```

1.9 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.

1.10 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 always starts with the key word *program* following with a program name.

Example: `Program name`

The following example shows how to call the function root in program m1 from the worksheet.

```
m1.root(9)=3
```

```
program m1

function root(a)
  result =  sqrt(a)
end
```

2.2 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
```

The expression can be a value, a variable, a data field, a function that returns a value or several of these. The example below defined ***x*** as a data field with 20 rows and 8 columns.

Example: `Define x[] = [1..20] * [1..8] fill NIL`

2.3 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"
Let x[5] = 16
```

The example above shows the last row assigned the value of 16 to index [5] of x. The first element is index [1].

2.4 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 = 1
While i < 100 do
  Statement Sequence....
  Let i = i + 1
End
```

2.5 If Then

The if *Then* 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
```

2.6 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
```

2.7 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
```

2.8 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.

Functions without arguments must have empty brackets behind the name.

Example: `Function Name()`

2.9 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
```

2.10 Call

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

The syntax of the *call* statement is:

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

2.11 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
```

2.12 Next

Use *Next* to assign a data series to a field variable. You can assign single value or single row to the field row by row. The special feature is that *next* needs no index. The field variables managed the index handling. Every execution of *Next* increments the index by one.

Example:
```
Define x = [1..20]

Next x
Next x = 23
Next x = 5.6
```

In the example above, *Next x* initialize the index of *x*. The next rows assign 23 to *x[1]* and 5.6 to *x[2]*. If you pass *x* as an argument to a function or another variable, the index will be pass on.

Another feature of *Next* is the range control. If the index reaches the tail of the data field, *Next* extend the data field automatically. However, for big data fields, it is useful to define the data field large enough. The late extension of the field needs more processing power. For small data fields up to few thousand records, this is irrelevant. In any case, it is important that you define a variable with a minimum of one record and the number of columns you need.

Next checks the data compatibility. The dimension of the assigned data field must be one less the field variable, or the same as one row of the field. The example

above assigned a simple value to a one-dimensional data field. In the example below, we move a one-dimensional field to a row of a two-dimensional field.

The numbers of columns can be different. The example below defined x as a two-dimensional field that contains 3 columns. In row 4 we assign a one-column field with the value of 99 to x. The remaining columns are filled with 0. In row 5 we assign a four-column field to *x*. *Next* cuts the fourth element.

Example:
```
Define x[] = [1..6]*[1..3] fill 1
Next x
Next x = [10,20,30]
Next x = [99]
Next x = [22,33,44,55]
```

Content of x:

10	20	30
99	0	0
22	33	44
1	1	1
1	1	1
1	1	1

2.13 Index

The function *Index* returns the current index of a field variable (the index of the last *Next* move).

Example:
```
i = Index(x)
```