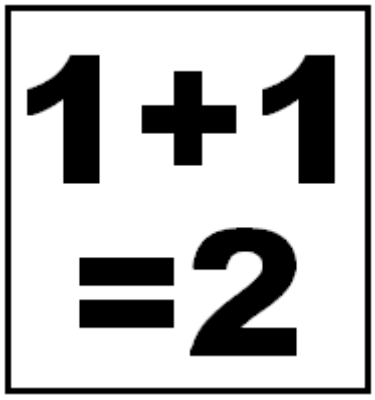
User Guide

For:
Rhyscitlema Calculator

By:
Rhyscitlema

http://www.rhyscitlema.com/applications/calculator



30 April 2018

Contents

1 General Information

Rhyscitlema Calculator is an application to evaluate complex expressions expressed in the *Rhyscitlema Function Expression Text* (RFET) language. RFET enables the representation of mathematical expressions in plain text in simple and effective ways. Instead of only thinking of a single-value, vector or matrix, the fully general **value-structure** is used: an example is (1,(2,3,4),5).

RFET enables evaluating simple expressions such as **1+1** , to evaluating advanced expressions such as **2*f(3); f(x)=4x** , to evaluating highly complex expressions in an Object-Oriented Programming model (using **inheritance** and **encapsulation**).

You can simply evaluate “4*5”. Now suppose you later want a few more of the form 4*X. In that case you evaluate say “4*(5, 6, 7, 8)”. Suppose you later want to also use 2 and 3 besides 4. In that case you evaluate say “f(2), f(3), f(4); f(x)=x*(5, 6, 7, 8)”. The result will be a 3-by-4 matrix. Suppose you later want to keep the current results but also evaluate with the vector (50, 61, 72, 83). In that case you evaluate “

```
f(2,a), f(3,a), f(4,a),
f(2,b), f(3,b), f(4,b) ;
f(x,y) = x*y ;
a = (5, 6, 7, 8) ;
b = (50, 61, 72, 83) ;
```

”. The result will be a 6-by-4 matrix.

The expression can get even more complex. But then you can always structure it effectively, thereby making evaluation a lot easier than ever! You can even make it vary with time and just watch, or make it be updated upon button presses. You can build an entire RFET script and save for later use.

RFET is designed and developed to be USERFOIL:

- **U**seful: can be used to do something important
- **S**imple: has as little different things as possible
- **E**fficient: provides high quantity+quality service
- **R**obust: survives unintended usage
- **F**lexible: gives full control to the user
- **O**riginal: not a cheap copy of some other
- **I**nteresting: definitely worth your time
- **L**asting: will survive the test of time

For now only technical information is provided.

Rhyscitlema products are provided for free, under the simple usage agreement: **use at your own risk**. In order to encourage improvement please consider donating at http://rhyscitlema.com/donate.

2 Rhyscitlema Function Expression Text (RFET)

A variable actually only serves the purpose of storing a value. This value is obtained from the evaluation of an expression provided in the form of the definition of a function. The user provides diverse function expressions in text in a way that makes it easy to express intended computations. This *way* essentially becomes known as the language which the user uses. Rhyscitlema provides yet another text-based language for function expressions called Rhyscitlema Function Expression Text.

Introducing the RFET language

An RFET is simply a block of text that evaluates to some result.

At the very beginning we have “1+1”, which is a basic math expression that evaluates to 2.
The expression can be made slightly more complex:

- 5-6/2+3*4
- 4*sin(3+2) + (5-2)^3^-4

Such a basic expression that only makes use of already defined features along with the needed operators is called an RFET statement, or simply a **statement**. A statement should be terminated by either the end-of-statement character **';'** (the semi-colon) or by the end of a text or end of a file.

The RFET block of text can be made even more complex:

1. 5a; a=4
2. 2*f(3); f(x)=4x
3. 5a+2*f(3); a=4; f(x)=4x

1. The 1st case evaluates 5a where a=4, so the result is 5*4 = 20.
2. The 2nd case evaluates 2*f(3) where f(x)=4x, so the result is 2*(4*3) = 24.
3. The 3rd case evaluates 5a+2*f(3) where a=4 and f(x)=4x, so the result is 20+24 = 44.

In these cases there are user-defined ***variables*** and ***functions***. Since they are user-defined there is therefore no restriction to only using software-defined variables and functions like *pi* and *sin()*.

Notice that a user-defined variable or function essentially defines a *statement*. That is for example a=4 is a variable ‘a’ evaluating the statement “4” while f(x)=4x is ‘f’ evaluating the statement “4x”.

Both a variable and a function are described as types of **components**. An RFET block of text is thus a collection of ***components definitions***, the first of which is directly an RFET statement. This very first statement is implicitly assigned to a special variable (or function as shall be seen) called ***main***.

The RFET block of text, which is a collection of components definitions, is also called a ***container***. A container can be inserted inside another container as an ***inner-container*** to a ***parent-container***. This inner-container will be itself a component sitting besides variables and functions components. As it is given a name, an inner-container can be called just like variables and functions are called.

3 RFET Component

Beyond the concept of an RFET statement (like for example “1+1”) is the concept of an RFET component. A component can be a variable, a function or an inner-container. Every component has a name. All components inside a ***containing container*** have unique names.

Consider the following example of two file contents showing statements, variables, functions, inner-containers, containing-containers and parent-containers:

For file1.rfet

```
1+1;
name = "file1.rfet";

\rfet{variable1;
    name = "container1";
    private variable1 = 10;
    enclosed variable2 = 20;
    protected variable3 = 30 + variable4;
    public variable4 = 40;
}
\rfet{variable2, variable4;
    type = "container1";
    name = "container2";
    public variable4 = 400;
}
```

For file2.rfet

```
2*f(3) ;
f(x) = 4x ;

type = "file1.rfet" ;
name = "file2.rfet" ;

\rfet{variable3 ,  "container2".variable3 ;
    type = "container1" ;
    name = "container3" ;
    replace variable4 = 4000 ;
}
```

3.1 As a variable and as a function

A component as a variable is defined in the form:

- variable_name = an_RFET_statement

A component as a function is defined in the form:

- function_name(parameter) = an_RFET_statement

A variable actually only serves the purpose of storing a value. For this reason a variable’s expression is evaluated to a constant value at parse-time. In future when sequential execution will (finally) be introduced it will become possible to change the value of a variable at run-time.

3.2 As a container – Parent vs Inner container

An RFET statement like “1+1” is the definition of a single component called the ***main*** component. An RFET container can be defined as a collection of components definitions, of which there is at least the *main* component’s definition, and optionally are variables, functions and inner-containers. Actually a variable or a function may be thought of as a container having only its *main* component.

An inner-container, as itself a component inside a containing container, is defined in the form:

- **\rfet{** *the_RFET_container_text* **}**

The classical components are variables and functions. The collection of all components defined within the same block forms a container. But this is yet considered to be itself a component (as an inner container) inside another container (its parent container). At the top-most level of this hierarchy is **the *Root* container.** The *Root* container has no variable nor function. It can only have inner-containers, the later which are said to be **top-level containers**.

The example with *file1.rfet* and *file2.rfet* illustrates multiple inner-components and their parent-containers. It should be observed that an RFET statement is always a *main* component’s definition.

A file that defines an RFET container is called an **RFET file**. It is typically loaded as a top-level container. It is recommended to **not** provide a *name* component in an RFET file. The file’s name will be detected and used implicitly upon loading it. But if however provided then this *name* component must always be the same as the file’s name.

A container can be called in the same way as a variable or function. The call calls the container’s *main* component, which is usually a variable but which can be defined as a function by using:

- \rfet{**\(parameter)=** *rest_of_the_RFET_container_text* }

Parent Container vs Containing Container

It is rather easy to get confused with the difference between a parent container and a containing container. It is all a matter of point of view. For a container *C* that looks at its inner-container *I* as just another one of its contained components, *C* is the containing-container of *I*, and is the parent-container to a variable/function found inside *I*. A variable/function is typically not thought of as being an inner-container itself. But if it was, then its containing-container will be thought of as the parent-container to its *main* component. Notice how the point of view changes. In summary, a parent-container is parent to a containing-container, not to a component definition.

3.3 OOP Inheritance – Base vs Derived container

The Object-Oriented Programming concept of inheritance is achieved using the ***type*** and ***name*** components of an RFET container. Both must be provided as direct strings.

A derived container is a container that inherits from a corresponding base container by setting its *type* to be equal to the *name* of that base container. The base container must be a ***sibling*** to the derived container and must have been defined before the derived container is defined.

Containers are considered to be siblings if they have the same parent or the parent of one inherits from the parent of another. In the example RFET provided, the inner-containers inside the files are all siblings because *file2.rfet* inherits from *file1.rfet*. *container1* is a ***direct sibling*** to *container2*

because they have the same parent *file1.rfet*. Meanwhile these 2 are ***indirect siblings*** to *container3* because *container3* is inside a different parent, *file2.rfet*, which inherits from *file1.rfet*.

If the container *type* is not specified then the container does not inherit anything. If the container *name* is equal to that of an indirect sibling (direct siblings cannot have the same name), and is thereby overriding that indirect sibling, then the *type* must also be equal to that name.

3.4 OOP Encapsulation – Component Access Control

The Object-Oriented Programming concept of encapsulation is achieved using a component access control mechanism that controls who can access the inner component of a containing container. This is done by preceding the component's definition by one of the 4 access control identifiers:

1. **private:** accessible only within its containing container (this includes inner-containers)
2. **enclosed:** accessible only by a container of **same parent** as its containing container
3. **protected:** accessible only by a container of **same grand-parent** as its containing container
4. **public:** accessible by anyone from anywhere

The example with *file1.rfet* and *file2.rfet* illustrates the component access control mechanism. Containers within an RFET file have that file as their containing container. Therefore a component with *enclosed* access will be accessible to a sibling within the file. RFET files are typically loaded as top-level containers, thereby sharing the *Root* container as their parent. Thus, inner-containers from these files will share the same grand parent, thereby having *protected* access level.

The access level of an overridden component cannot be downgraded. That is if a derived container overrides a component of a base container, then the access level of the new component definition (found inside the derived container) must not be less than that found inside the base container.

Important:

1. When not specified the access type is made ***protected*** **by default**.

2. Top-level containers are always set to have ***public*** access type.

3. The *name* component is always set to ***private*** access irrespective of what the user specifies.

4. Upon moving up the hierarchy, that is from inner to parent container, the access level does not change. So the call **"..|..|..|.."."var** will access *var* even if *var* has private access.

5. Upon moving down the hierarchy, that is from parent to inner container, the access level increases by 1, and increases no more when it becomes *public* access. So a parent container can access its inner-container's component only if the later has at least *enclosed* access level.

A certain feature that acts like an access control, although it is technically not one, is the ***replace*** component identifier. A component inside a derived container is marked as *replace* provided that it **overrides** an existing component found inside a base container and for which it can access.

The purpose of the *replace* identifier is for a derived container to tell the target base container that any usage of that **replaced** component must be made using the replaced component definition. In the example RFET provided, *container3* evaluates *variable3* while haven replaced *variable4*. So the result will be 30+4000 = 4030. It could not have used *variable2* because it cannot access it.

The *replace* identifier behaves as an access control with access level **5**, which is higher than *public*. This is so that a component marked as *replace* cannot be overridden by a derived container into a component not marked with as *replace* (this aim certainly can be achieved in a different way!)

3.5 Calling a Component – Relative vs Absolute path

A useful feature is the ability to call any component inside an RFET statement. This is done using:

- "path".name

path is the path to the container that contains the component of name ***name***.

A path that starts:

- with '|' , implying an **absolute path**, starts the search from the *Root* container and with *public* access level.
- with '.' , which denotes **current**, starts the search from the *containing* container and with *private* access level.
- with '..', which denotes **parent**, starts the search from the *parent* container and with *enclosed* access level.
- otherwise, the search **defaults** to starting at the *parent* container, again and with *enclosed* access level.

In the example RFET provided, the call *"container2".variable3* starts the search from the parent container *file2.rfet*. If the call was instead *"**.**|container2".variable3* then the search would have started from the containing container *container3*.

Note that the call *"path".**main*** is always valid even for variables and functions, since every component is a container containing at least this *main*-component. So even the call *"path|var|main|main".main* is valid, and will resolve to the exact same as doing *"path".var* .

Note: the call *"|filename".component* will automatically load the RFET file if not already loaded.

3.6 GLOBAL.RFET top-level container

Software-defined variables and functions such as *e*, *pi*, *sin* and *cos* are characterised by the property that they can be called from anywhere. There is no concept of inheritance involved.

In order to provide this same property to user-defined variables and functions the concept of **globally accessible components** is used. Basically, all globally accessible user-defined components must be defined inside a special top-level container with name = "GLOBAL.RFET".

For example, evaluating the below will make *my_cos* to become globally accessible for future use:

```
0;
name = "GLOBAL.RFET" ;
my_cos(x) = cos(2*x) ;
```

One may consider it convenient to define globally accessible components inside a file with name GLOBAL.RFET and then load this file directly after launching the software.

3.7 Evaluating the given example

The example RFET provided can be loaded by evaluating the first file's content (which will result to 2 due to "1+1") and then evaluating the second file's content (which will result to 24). The RFET files' contents will load as top-level containers. Everything will load properly, except that there will be no useful result to observe and play with! Below is a single container that takes in the two files' contents as inner-containers. Evaluating it will produce useful results to observe and play with.

```
".|file1.rfet".container1 ,
".|file1.rfet".container2 ,
".|file2.rfet".container3 ,
".|file2.rfet".main ,
".|file1.rfet".main ;

\rfet{
    1+1;
    name = "file1.rfet";

    \rfet{variable1;
        name = "container1";
        private variable1 = 10;
        enclosed variable2 = 20;
        protected variable3 = 30 + variable4;
        public variable4 = 40;
    }
    \rfet{variable2, variable4;
        type = "container1";
        name = "container2";
        public variable4 = 400;
    }
}
\rfet{
    2*f(3) ;
    f(x) = 4x ;

    type = "file1.rfet" ;
    name = "file2.rfet" ;

    \rfet{variable3 , "container2".variable3 ;
        type = "container1" ;
        name = "container3" ;
        replace variable4 = 4000 ;
    }
}
```

The result of evaluation is:

```
(10,
 (20, 400),
 (4030, 70),
 24,
 2)
```

4 RFET Statement – syntax

The concept of an RFET component has been discussed. A variable or function is defined using an RFET statement. The following sub-sections discuss the syntax of an RFET statement in details.

4.1 Value-Structure

The value-structure defines the structure of the result of an evaluation of an expression.
Particularly:

- A value structure is represented using brackets, comma-separators and names.
  - ((w,x,y),z) {w,x,{y,z}}
- A single value is the most common value structure
  - x , (x) , ((x))
- Vectors and matrices are special types of value structures
  - (w,x,y,z) # a 4-values vector
  - ((w,x), (y,z)) # a 2-by-2 matrix

- The parameter to a function is defined using a value structure
  The brackets **()** must be used
  - f((x,y),z) = x+y+z ;

- The brackets pairs {} and () are always interchangeable. The only exception is that () is used when specifying a function parameter or specifying a function call argument.

4.2 Supported Operators

All the available operators are listed below. The order in which they are listed is also their **precedence order**. That is what comes lastly is evaluated firstly. For example, because the operator times * comes after the operator plus +, an expression such as 2+3*4 will evaluate as (2+(3*4)). However some have equal precedence, such as times and divide. *Do not rely on these; use brackets wherever possible*. A single-value is considered as *false* if it evaluates to zero (or empty string) and as *true* otherwise. The concept of *per-value operation* is mentioned here but is explained later.

- open-close brackets **( )** and **{ }**
- brackets for subscripting **[ ]**
- replacement operator **:=**
- comma separator **,**
- mix/combine **.,**
- conditional **? :**

- Logical OR **or** (**these** do only per-value operations)
- Logical AND **and**
- Logical NOT **not**

- Identical to **===** (*never* does per-value operation, always returns *true* or *false*)
- Equal to **==** (**these** do only per-value operations)
- Not equal **!=**
- Less than **<**

- Greater than >
- Less or equal <=
- Greater or equal >=

- Plus + (does only per-value operation)
- Minus - (does only per-value operation)
- Positive +
- Negative -
- Times ***** (or **.*** for per-value operation)
- Divide / (or **./** for per-value operation)
- To-power ^ (or **.^** for per-value operation)
- Integer divide // (does only per-value operation)
- modulo|remainder **mod** (does only per-value operation)

- Bitwise *right shift* >> (**these** do only per-value operations)
- Bitwise *left shift* <<
- Bitwise *xor* ^|
- Bitwise *or* |
- Bitwise *and* **&**
- Bitwise *not* ~

- Dot Product • (Unicode character \u2022)
- Transpose **^T**

4.3 Software-Defined Constants

- booleans **true** **false**
- e constant **e**
- pi constant **pi**
- square root of -1 **i**
- full path to the container **this** (a string)

4.4 Software-Defined Math Functions

- greatest common divisor **gcd(n, m)**
- integer logarithm base b **ilog(b, n)**
- integer square root **isqrt()**
- floor **floor()**
- ceiling **ceil()**
- factorial **factorial()** (for a real number x, result = ***gamma_function(x+1)***)

- e raised to power **exp()**
- logarithm base e **log()**
- square root **sqrt()**
- cube root **cbrt()**

- complex number magnitude **cabs()**
- complex number argument **carg()**

- complex number real part **real()**
- complex number imaginary part **imag()**
- complex number conjugate **conj()**
- complex number projection **proj()**

- trigonometric sine **sin()**
- trigonometric cosine **cos()**
- trigonometric tangent **tan()**
- trigonometric sine inverse **asin()**
- trigonometric cosine inverse **acos()**
- trigonometric tangent inverse **atan()**

- hyperbolic sine **sinh()**
- hyperbolic cosine **cosh()**
- hyperbolic tangent **tanh()**
- hyperbolic sine inverse **asinh()**
- hyperbolic cosine inverse **acosh()**
- hyperbolic tangent inverse **atanh()**

4.5 Software-Defined Extra Functions

- convert to string **tostr()**
- convert to number ~~n/a~~
- convert to big number ~~n/a~~
- convert to small number ~~n/a~~

- minimum of all single-value elements **min()**
- maximum of all single-value elements **max()**
- summation of all single-value elements **sum()**
- count of all single-value elements **span()**

- size of a vector or matrix **size(*v*)**
  - result is the 2-value vector: (rows of *v*, columns of *v*)
  - if *v* is neither a vector nor a matrix then columns = 0

- get a vector in given range **range(start, increment, stop)**
  ex: range(1,(4,2),50) = (1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49)

- get a vector of given length **vector(start, increment, length)**
  ex: vector(1,(4,2),17) = (1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49)

- get prime factorisation ~~n/a~~
- get prime numbers in range ~~n/a~~
- ~~**fullfloor()**~~ , see http://rhyscitlema.com/algorithms/math-fullfloor-function

- try and catch expressions **try(this, that, ..., catch)**
  Evaluate the expression *this*. If no error then return the result to the caller. Else evaluate the expression *that*, if error then keep evaluating the next one up until the last one called *catch*.

- **print(**expression**)** , evaluate expression and print the result as an error message.

- **eval(**"rfet"**)** , evaluate expression in given string and return the result to caller.
- **eval(**"\\(*parameter*)= rfet"**,** *argument***)** , evaluate given string as a function
  ex: eval("\\(x,y)=x*y", (2,3)) = 6

4.6 The call() function

This is used to conveniently call a user-defined function with **customised** argument passing:

- **call(***path***,** *mask***,** *arg0***,** *arg1***,** *arg2***, ...,** *argN***)**

mask = a positive integer used as described below.
path = path to the function to call given as a string.

The user-defined function to call must take N+1 arguments from arg0 to argN. The arguments to this function are provided such that the **rows (first-level elements)** of arg*X* are passed in place of arg*X* itself **only if** (*mask* & (1<<*X*)) != 0 , otherwise arg*X* itself is passed.

Consider the example below:

```
f, a, b;
a =  f(n, 0, 3) , f(n, 1, 4) , f(n, 2, 5) ;
b = call( f, 0b110, n, (0,1,2), (3,4,5) ) ;
n = (6, 7, 8, 'A', 'B', 'C') ;
f(x, i, j) = x[i], x[j] ;
```

Result of evaluation is:

```
(|entry|f,
 ((6, A), (7, B), (8, C)),
 ((6, A), (7, B), (8, C)))
```

The result shows that both variables *a* and *b* evaluate to the same thing. Actually the expression for variable *a* is a ***demonstration*** of how exactly the evaluation of the expression for variable *b* is done.

Therefore it can be observed that since (*mask* & (1<<0)) == (0b110 & 1) == 0 , the variable *n* which corresponds to *arg0* is provided directly as argument to the function *f* being called, while *arg1* and *arg2* are not provided directly but instead it is their respective **rows** that are provided as argument.

call("*function*", *argument***)**	is the same as call("*function*", 1, *argument*)	where mask = 1
call("*variable*"***)***	is the same as call("*variable*", 0)	where mask = 0

4.7 Replacement Operator :=

Consider the following expression:

- 0 := LHS + 1

The replacement operator **:=** evaluates what is on its right-hand-side (RHS), then replaces what is on its left-hand-side (LHS) with the result. ***LHS*** is a special variable considered special **only when** there is an associated replacement operator. Using it causes the replacement to be an update.

4.8 Conditional Operators " ? : "

The pair of conditional operators " ? : " enables the use of conditional expressions. This means with this pair it is possible to do piece-wise definition of variables and functions.

The syntax is: **(condition) ? (on true) : (on false)**
which is equivalent to the piece-wise definition below:

- = (on true) if (condition) or
  (on false) otherwise

If *condition* evaluates to:

- *true* then (on true) is evaluated and (on false) is ignored
- *false* then (on true) is ignored and (on false) is evaluated

For example consider the expression: 2 + ((5>6) ? 1 : -1)
The result is evaluated as 2 + (-1) because the expression (5>6) evaluates to false.

The two operators have the same precedence level. So an expression like:

- 2 + ((5>6) ? 1 : (5<6) ? 0 : -1)

which can be rewritten as

- 2 + ((5>6) ? 1 :
  (5<6) ? 0 : -1)

will evaluate as

- 2 + 1 if (5>6) or
  0 if (5<6) or
  -1 otherwise

4.9 Per-value Operation

Some operators are described to do **per-value operation**. This means the operation is applied to each of the corresponding values/elements of the value-structures of the operands. For example the plus + operator does per-value operation. For example:

- (1, 2, 3) + 4 evaluates as (1+4, 2+4, 3+4)
- (1, 2, 3) + (4, 5, 6) evaluates as (1+4, 2+5, 3+6)

An operation that normally can only be performed on a single-value argument, like for example the Math functions, will do *per-value operation* if the operand contains more than one value. Example:

- sin(1, 2, 3) evaluates as (sin(1), sin(2), sin(3))

4.10 String / Text Expression

A string (or a text expression in general) is anything found inside the pair of double quotes " ". It is a type of **single-value**. The result of evaluating (2+3,"text") is (5, text). Supported operations are:

- To obtain the length of a string: strlen("abcd") = 4
- Indexing a string gives a character: "abcd"[0] = 'a'
- Slicing a string gives a string: "abcd"[1, 2] = "bc"
- Slicing is done same as in *vector slicing*, except that: "abcd"[-1] = '\0'
- Slicing is used to reverse a string (see *vector slicing*): "abcd"[-2, -1, 0] = "dcba"
- A value + a string is converted to a string then added: (10+2)+"%" = "12%"

4.11 Vector Indexing and Slicing

Vector indexing and slicing is done using the syntax *operand*[*arg*]. They operate only on the ***rows (first-level elements)*** of the operand. When *arg* is a single integer *i* the operation is indexing. When *arg* is the 3-integers vector (*start*, *incre*, *stop*) slicing is done where *start* is the starting index, *stop* is the **inclusive** stopping index and *incre* is the index incrementation value. When *arg* is the 2-integers vector (*start*, *stop*) slicing is done with an implicit value of *incre* = 1. Examples:

- (5,6,7,8)[0] = 5
- (5,6,7,8,9)[1,3] = (6,7,8)
- (4,5,6,7,8,9)[1,2,5] = (5,7,9)

Any other value structure of *arg* is invalid. Note that **any integer is valid** including negative integers. Precisely the given integer (say *i*) is first converted so that it fits within the range **[0,*r*]** (where *r* is the number of rows): fit(*i*,*r*) = f(*i*,*r+1*); f(*i*,n) = g(*i* mod n, n); g(j,n) = (j<0 ? j+n : j);
Note: indexing at [*r*] gives the **empty vector**. Therefore (5,6,7,8)[-1] = (5,6,7,8)[4] = ( ) .

4.12 Commenting

The character **#** is used to comment a single line. That is everything from the character up to the encounter of a newline is ignored, except that newline.

#{ and **}#** are used for *block-commenting*. That is everything from the specific starting point to the specific stopping point is ignored. A }# only closes a matching #{. Therefore **a block-comment can contain another block-comment**.

Priority is given to the first encountered commenting. That is for example: #...#{ or #...}# on a single line, is a valid and single-line comment instance – the line is commented. Also #{...#...}# is a valid and single block-comment instance – the block of text is commented.

4.13 More about RFET

- By default a given vector say (0,0,0) for example, is a **column** vector.

- A function parameter overrides any component (that is any user-defined variable or function or container), the later which in turn overrides any software-defined variable or function.

- Recursive function call is possible.
  e.g: " *fib(10); fib(n) = (n<=1) ? n : fib(n-1) + fib(n-2);* " = 55
  e.g: " *gcd(35, 65); gcd(a, b) = (a==0) ? b : gcd(b mod a, a) ;* " = 5

- The value structure can change during evaluation.
  e.g: " f(1), f(2,3); f(x) = x+x; " = (2, (4, 6))

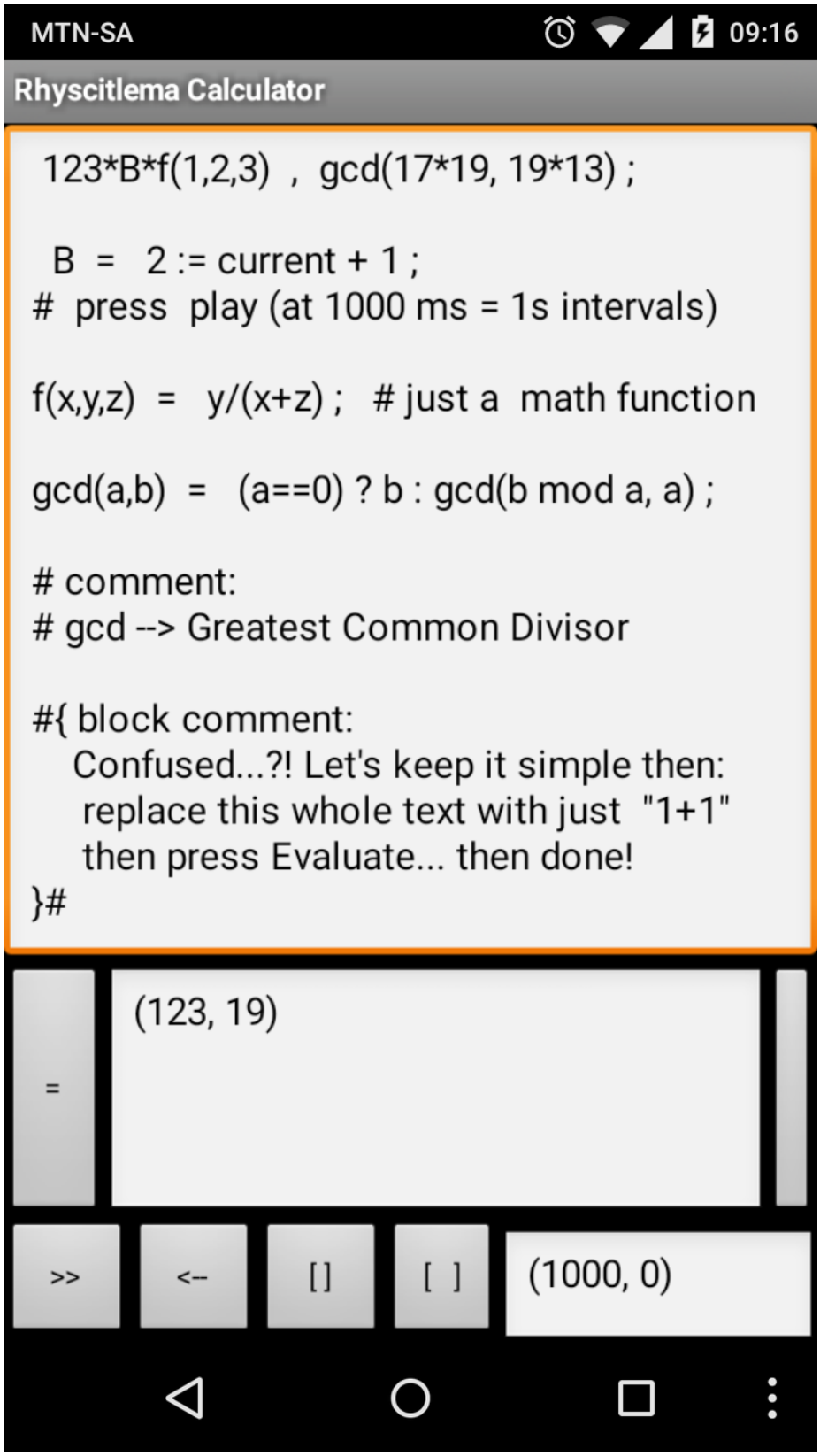
- An example given number in base 2 is **0b**101.01 , in base 8 is **0o**273.14 , in base 16 is **0x**FEA.8D . A number can be given in an arbitrary base B as: ***tonum(w,x,B,y,z)*** , where w,x,y,z are the digits in the interval [0, B), and B denotes the position of the decimal point.

- ***tostr(base, (value structure))*** converts the value structure into a string in the given base.
  e.g: " tostr(2, (4,5,6)) " = (0b100, 0b101, 0b110)

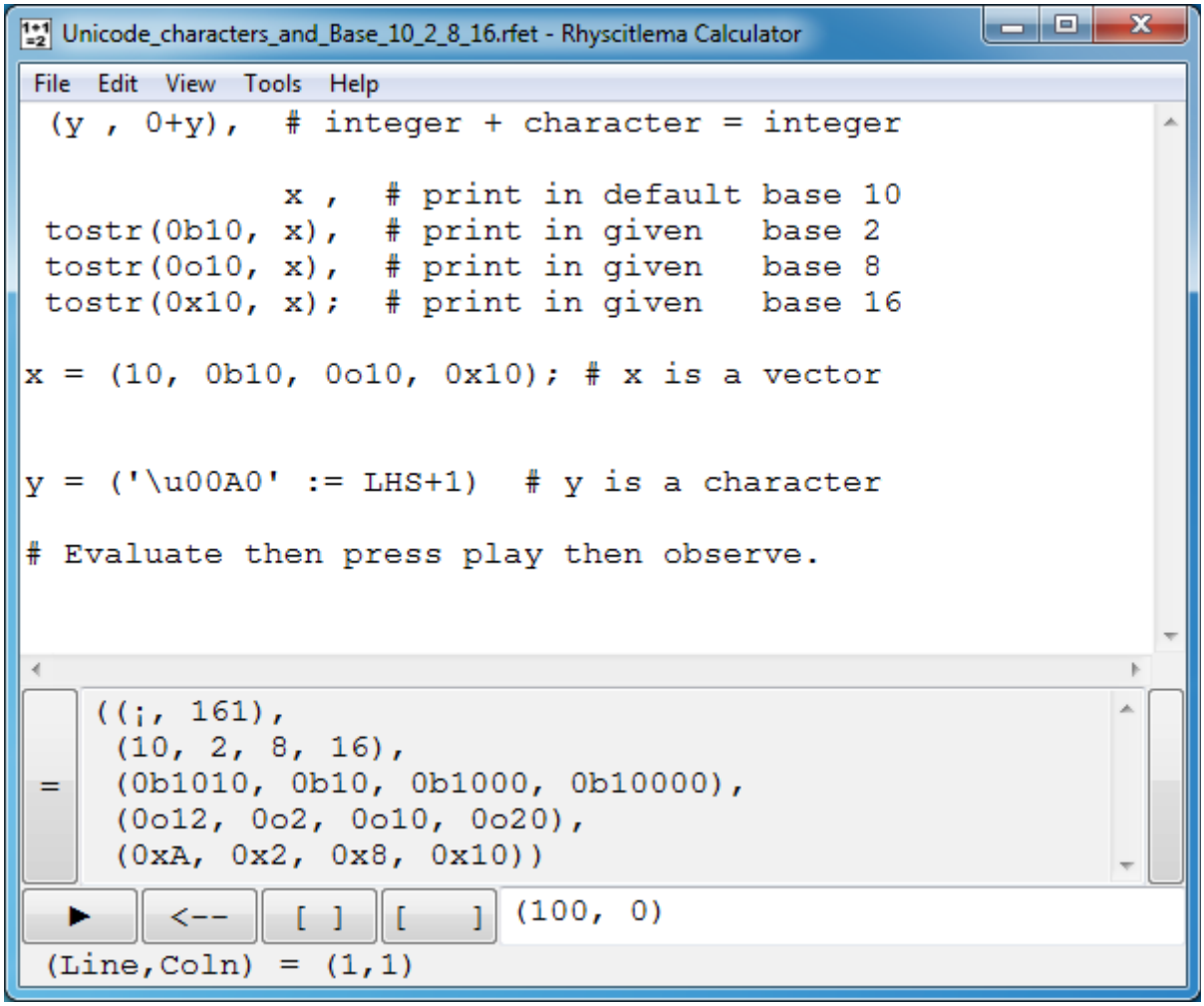
- The *combine/mix* operator **.,** mixes **first-level** elements. e.g: " (1,2) ., (4,5) " = (1,2,4,5)
- Changes made to an inner-container ***reflects*** in all its ancestor containers.

5 Graphical User Interface

5.1 GUI on Android



5.2 GUI on Windows



The user interface largely does text and file editing. Features that are common to most text file editing software will not be discussed. When a file with the .rfet (or .rodt) extension is loaded its content is displayed in the large entry text field and then automatically evaluated. The result of evaluation is displayed in the wide message text field found in the middle of the GUI's 1st layer.

5.3 Software-specific Menu features

- Menu → Edit → Go To:
  - Repositions the caret. The provided entry is an RFET.

- Menu → Edit → Convert Text:
  - These are features to choose whether to show a text in its normal character form, partly-code-number form, or fully-code-number form. This is explained in details later.

- Menu → Tools → Evaluate:
  - Does the same thing as the *Evaluate* (=) button (explained later).

5.4 1st layer buttons and text field

The *Evaluate* (=) button evaluates the content of the large entry text field as an RFET entry (it evaluates the *main* component), then displays the result in the wide message text field found in the middle of this layer. Evaluation can also be performed by using the key combination **Shift+Enter**.

The evaluation may commit further changes to the software depending on what has been evaluated. For example if an RFET container of type UIDT (User Interface Definition Text) is evaluated then this container will be used to update the GUI of the software. For example evaluating the below will increase the height of the 1st layer of the GUI:

```
0;
type = "User_Interface_Definition_Text" ;
name = "My Custom GUI" ;
replace h1 = 150 ;
```

The ***Lock*** button on the far right is used to get keyboard input. Consider for example:

```
0 := LHS + (Key_a or Key_A) - (Key_b or Key_B) ;
```

Then after selecting the *Lock* button, pressing the character ‘a’ or ‘A’ will increment the left-hand-side of the replacement operator, meanwhile pressing the character ‘b’ or ‘B’ will decrement it.

5.5 3rd layer buttons and text field

At the far right of the 3rd layer is the time text field that evaluates the RFET: *(TimerPeriod, time)*. Both are software-defined variables. *TimerPeriod* is the periodic interval (in milliseconds) after which the value of *time* is increased by *TimerPeriod/1000* seconds.

The user entry is **re-evaluated** again each time the *time* variable is updated. Pressing the *Play/Resume* button ‘►’ will start the timer. The button will then switch to the *Pause/Suspend* button ‘||’, the later which when pressed will stop the timer.

Pressing the *Reverse/Forward/Backward (← or →)* button reverses the sign of the timer period. If *TimerPeriod* is negative then the *time* variable will decrease on every update. Pressing the *LowerPeriod* button ‘[ ]’ will reduce the timer interval by an amount specified in the User Interface Definition Text (UIDT). The default reduction is 25 milliseconds. Pressing the *HigherPeriod* button ‘[ ]’ will increase the timer interval in the same way. Note however that for all these cases, as the value of time does not change the user entry is not re-evaluated.

The time text field can be updated directly by editing the text then pressing the *Done/Enter* key.

5.6 Hidden layers

There are more GUI features, such as the 2nd and 4th layers of buttons and text fields. These are only discussed in the user guide for the Rhyscitlema Graph Plotter 3D software. They are hidden in the Rhyscitlema Calculator software because of being rarely needed. However the functionalities of the 2nd layer buttons are made available through **Menu→Tools**:

- *Prev* and *Next* container: are both used to scroll through a list of all containers.
- *Delete* container: deletes the selected container (as specified in the path text field).
- *Clear* container path: is used to clear the path text field, meaning no container is selected.

6 Character vs Partly-Code-Number Text Display

This is a feature for file editing. It is especially useful to:

- Efficiently use the Find-Replace user interface feature
- Provide a character through its code number
- Know the code number associated to a character

A character is stored in a computing device in the form of a number. The displayed character is just the meaning given to that number. For example the character '0' is stored as the hexadecimal number 30 (or 0x30), corresponding to the decimal number 3*16+0 = 48. This is the *Code Number*.

Here, a character with a 4-digit hexadecimal code number of XXXX is represented as **\uXXXX**. For example the character '0' is represented as \u0030. This is described as the fully-code-number representation.

In certain contexts such as software programming, some characters have alternative representations. Particularly:

- The tab character is represented as **\t** (code number \u0009)
- The carriage-return character is represented as **\r** (code number \u000D)
- The line-feed or newline character is represented as **\n** (code number \u000A)

In regard to this feature of the user interface, a character that is said to be displayed as partly-code-number is a character displayed in one of the following forms:

- The **normal character display**
  - The '0' character will display as '0'
  - The newline character will display a new line

- The **alternative representation** if it is available
  - The '0' character will display as '0' (since it has no alternative representation)
  - The newline character will display as \n (and there will be no actual new line)

- The **fully-code-number representation**
  - The '0' character will display as \u0030
  - The newline character will display as \u000A

The feature “Set Partly is Fully Code Number” is a feature to redefine the partly-code-number representation of a character, so that using the feature “Character to Partly Code Number” will keep most characters unchanged while showing the fully-code-number representation of the affected characters only. To use this feature a text is first selected, which contains the targeted characters in their fully-code-number representation.

These features can be applied to any selected text, including texts found in the *find-replace* dialog boxes – and yes this, is the original reason why these features were developed in the first place!

7 Few Simple Examples

7.1 Basic Example

RFET:

```
   123 * β * ƒ(1,2,3) ;

 β = 2 ;  ƒ(x,y,z) = y/(x+z) ;
```

Result of evaluation:

```
123
```

7.2 Circles and Spheres

RFET:

```
  Result ;

 r = 1 ;            # radius
 C = 2*π*r ;        # Circumference of a circle
 A = π*r^2 ;        # Area of a circle
 S = 4*π*r^2 ;      # Surface of a sphere
 V = (4/3)*π*r^3 ; # Volume of a sphere
 π = pi ;           # 'pi' is software-defined

 Result = (C , A) , (S , V) ;
```

Result of evaluation:

```
((6.283, 3.142),
 (12.566, 4.189))
```

7.3 Quadratic Equation

RFET:

```
  x1 , x2 , f(x1) , f(x2) ;

 x1 = ( -b + sqrt(b^2 - 4*a*c) ) / (2a) ;

 x2 = ( -b - sqrt(b^2 - 4*a*c) ) / (2a) ;

 f(x) = a x^2 + b x + c ;

 a = 2; b = -2;  c = 1;

 # Solving for x where f(x) = 0
 # Change the coefficients a, b and c
```

Result of evaluation:

```
(0.5+0.5i, 0.5-0.5i, 0, 0)
```