GTL Template language

Jean-Luc Béchennec

August 10, 2016

Contents

1 Data types

GTL supports the following data types:

int arbitrary precision integer numbers. The GMP library is used;

float 64 bits floating point numbers;

bool standard boolean;

type the type of a data;

string unicode strings;

struct structured data;

list lists of data, may be accessed as a table;

map map (aka dictionary) of data;

unconstructed an unconstructed variable.

Each type has its set of operators, getters and setters. The expression, for getters, or the variable, for setters, is called the *target*. Getters return a value related to the data but do not change it. They are used to get information from a data or to convert it into another type. Setter may target a literal expressions. Setters change the data and do not return anything. Setter may only target a variable. Getters and setters may have arguments. Syntax for getters without argument is as follow:

```
[expression getter]
```

When the getter takes arguments, they are listed after a colon and separated commas as follow:

```
[expression getter : arg1, arg2, ..., argN]
```

Syntax for setters without arguments is as follow:

```
[!variable setter]
```

When the setter takes arguments, they are listed after a colon and separated by commas as follow:

```
[!variable setter : arg1, arg2, ..., argN]
```

1.1 getters applicable to any data type

`type` returns the data type of the expression. See the section **??**.

`isANumber` returns **`true`** if the expression is a number: `int` or `float`, **`false`** otherwise.

1.2 `int` data type

The `int` data type support arbitrary precision arithmetic by using the GNU Multiple Precision Arithmetic Library (GMP).

1.2.1 `int` operators

The `int` datatype supports the following operators:

Unary operators

Operator	Expression type	Meaning
+	int ← +int	Plus operator. No effect
-	int ← -int	Minus operator. Negation
~	int ← ~int	Not operator. Complementation by 1

Binary arithmetic operators

Operator	Expression type	Meaning
+	int ← int + int	Addition
-	int ← int - int	Substraction
*	int ← int * int	Multiplication
/	int ← int / int	Division
mod	int ← int mod int	Modulus

Binary bitwise operators

Operator	Expression type	Meaning
&	int ← int & int	bitwise and
|	int ← int | int	bitwise or
^	int ← int ^ int	bitwise exclusive or
<<	int ← int << int	shift left
>>	int ← int >> int	shift right

Comparison operators

Operator	Expression type	Meaning
!=	bool ← int != int	Not equal
==	bool ← int == int	Equal
>	bool ← int > int	Greater than
<	bool ← int < int	Lower than
>=	bool ← int >= int	Greater or equal
<=	bool ← int <= int	Lower or equal

1.2.2 `int` getters

getter	Type	Meaning
`string`	string	Returns a string decimal representation of the int expression. `[42 string]` returns string `"42"`.
`hexString`	string	Returns a string hexadecimal representation of the int expression prefixed by `0x`. If the expression is negative a '-' is inserted before. `[42 hexString]` returns string `"0x2A"`. `[-1 hexString]` returns string `"-0x1"`.
`xString`	string	Returns a string hexadecimal representation of the int expression. If the expression is negative a '-' is inserted before. `[42 xString]` returns string `"2A"`. `[-42 xString]` returns string `"-2A"`.
`numberOfBytes`	int	Returns the number of bytes needed to store an unsigned expression. `[255 numberOfBytes]` returns `1`, `[256 numberOfBytes]` returns `2`
`signedNumberOfBytes`	int	Returns the number of bytes needed to store a signed expression. `[127 numberOfBytes]` returns `1`, `[128 numberOfBytes]` returns `2`
`numberOfBits`	int	Returns the number of bits needed to store an unsigned expression. `[63 numberOfBits]` returns `6`, `[64 numberOfBits]` returns `7`
`signedNumberOfBits`	int	Returns the number of bits needed to store a signed expression. `[63 signedNumberOfBits]` returns `7`, `[64 signedNumberOfBits]` returns `8`
`sign`	int	Returns `-1` if the expression is strictly negative, `0` if it is null and `+1` if the expression is strictly positive.
`fitsUnsignedInByte`	bool	Returns `true` if the expression fits in an unsigned byte, `false` otherwise.
`fitsSignedInByte`	bool	Returns `true` if the expression fits in a signed byte, `false` otherwise.
`fitsUnsignedInWord`	bool	Returns `true` if the expression fits in an unsigned 16 bits word, `false` otherwise.
`fitsSignedInWord`	bool	Returns `true` if the expression fits in a signed 16 bits word, `false` otherwise.
`fitsUnsignedInLong`	bool	Returns `true` if the expression fits in an unsigned 32 bits long, `false` otherwise.

getter	Type	Meaning
`fitsSignedInLong`	bool	Returns `true` if the expression fits in a signed 32 bits long, `false` otherwise.
`fitsUnsignedInLongLong`	bool	Returns `true` if the expression fits in an unsigned 64 bits long long, `false` otherwise.
`fitsSignedInLongLong`	bool	Returns `true` if the expression fits in a signed 64 bits long long, `false` otherwise.
`abs`	int	Returns the absolute value of the expression.
`bitAtIndex`	bool	This getter takes one argument: `index`. It returns `true` if the bit at index `index` is set and `false` otherwise. `index` 0 corresponds to the lowest significant bit. `[1 bitAtIndex: 0]` returns `true`

1.2.3 `int` setters

setter	Meaning
`setBitAtIndex`	This setter takes two arguments. The first one, `value`, is a bool. The second one, `index`, is the index of the bit to set. if `value` is `true` the bit is set to `1` and to `0` otherwise. Assuming `a` contains `0` at start, `[!a setBitAtIndex: true, 0]` sets `a` to `1`.
`complementBitAtIndex`	This setter takes one argument, `index`, which is the index of the bit to complement. Assuming `a` contains `1` at start, `[!a complementBitAtIndex: 1]` sets `a` to `3`.

1.3 The `float` data type

The `float` data type is the standard IEEE784 64 bits floating point number.

1.3.1 `float` operators

The `float` data type supports the following operators:

Unary operators

Operator	Expression type	Meaning
+	`float` ← `+float`	Plus operator. No effect
-	`float` ← `-float`	Minus operator. Negation

Binary arithmetic operators

Operator	Expression type	Meaning
+	`float` ← `float + float`	Addition

Operator	Expression type	Meaning
-	float ← float - float	Substraction
*	float ← float * float	Multiplication
/	float ← float / float	Division

Comparison operators

Operator	Expression type	Meaning
!=	bool ← float != float	Not equal
==	bool ← float == float	Equal
>	bool ← float > float	Greater than
<	bool ← float < float	Lower than
>=	bool ← float >= float	Greater or equal
<=	bool ← float <= float	Lower or equal

1.3.2 `float` getters

getter	Type	Meaning
string	string	Returns a string representation of the float expression. [4.2 string] returns string "4.2".
cos	float	Returns the cosine of a float expression expressed in radian.
sin	float	Returns the sine of a float expression expressed in radian.
tan	float	Returns the tangent of a float expression expressed in radian.
cosDegree	float	Returns the cosine of a float expression expressed in degree.
sinDegree	float	Returns the sine of a float expression expressed in degree.
tanDegree	float	Returns the tangent of a float expression expressed in degree.
exp	float	Returns the exponentiation of a float expression.
logn	float	Returns the natural logarithm of a float expression.
log2	float	Returns the logarithm base 2 of a float expression.
log10	float	Returns the logarithm base 10 of a float expression.
sqrt	float	Returns the square root of a float expression.
power	float	This getter takes one argument, p. It returns the expression raised to the power of p.

1.4 The `string` data type

The `string` data type supports unicode. A literal string is delimited by a pair of ". Literal strings support special characters:

Escape sequence	Corresponding character
\f	form feed
\n	new line
\r	return

Escape sequence	Corresponding character
`\t`	`horizontal tab`
`\v`	`vertical tab`
`\\`	`backslash`
`\Ø`	`null character`
`\u`*nnnn*	`unicode character with code` *nnnn* `in hexadecimal`
`\U`*nnnnnnnn*	`unicode character with code` *nnnnnnnn* `in hexadecimal`

1.4.1 `string` operators

The `string` data type supports the following operators:

Binary operator

Operator	Expression type	Meaning
`+`	`string` ← `string + string`	Concatenation

Comparison operators

Operator	Expression type	Meaning
`!=`	`bool` ← `string != string`	Not equal
`==`	`bool` ← `string == string`	Equal
`>`	`bool` ← `string > string`	Greater than
`<`	`bool` ← `string < string`	Lower than
`>=`	`bool` ← `string >= string`	Greater or equal
`<=`	`bool` ← `string <= string`	Lower or equal

1.4.2 `string` getters

getter	Type	Meaning
`HTMLRepresentation`	string	Returns a representation of the string suitable for an HTML encoded representation. '&' is encoded by & , '"' by " , '<' by < and '>' by > .

getter	**Type**	**Meaning**
`identifierRepresentation`	string	Returns an unique representation of the string conforming to a C identifier. Any Unicode character that is not a latin letter is transformed into its hexadecimal code point value, enclosed by `'_'` characters. This representation is unique: two different strings are transformed into different C identifiers. For example: `value3` is transformed to `value_33_`; `+=` is transformed to `_2B__3D_`; `An_Identifier` is transformed to `An_5F_Identifier`.
`fileExists`	bool	Returns `true` if a file exists at the target path, `false` otherwise.
`length`	integer	Returns the number of characters in the string
`lowercaseString`	string	Returns the lowercased representation of the string.
`capitalized`	string	if the string is empty, this getter returns the empty string; otherwise, it returns the string with the first character being replaced with the corresponding upper case character.
`uppercaseString`	string	Returns uppercased representation of the receiver
`leftSubString`	string	Returns the sub-string from the beginning of the target and with the number of characters passed as argument. If the sub-string is longer that the target, the target is returned. `["Hello World !" leftSubString : 5]` returns `"Hello"`.
`rightSubString`	string	Returns the sub-string from the end of the target and with the number of characters passed as argument. If the sub-string is longer that the target, the target is returned. `[["Hello World !" leftSubString : 11] rightSubString: 5]` returns `"World"`.

getter	Type	Meaning
`subString`	string	Returns the sub-string from the `index` passed as first argument and with the number of characters passed as second argument. If the `index` is out of the target, the empty string is returned. If the number of characters is greater than the sub-string, the sub-string is returned. `["Hello␣World␣!" subString : 6, 5]` returns `"World"`. `["Hello" subString : 10, 3]` returns the empty string. `["Hello" subString : 2, 10]` returns `"llo"`.
`reversedString`	string	Returns a mirrored string. `["Hello␣World␣!" reversedString]` returns `"!␣dlroW␣olleH"`.
`componentsSeparatedByString`	list	This getter takes one string argument: `separator`. The target is cut into pieces according to the separator and a list of the pieces is returned. `["Hello␣World␣!" componentsSeparatedByString : " "]` returns `@( "Hello", "World", "!" )`.
`columnPrefixedBy`	string	This getter takes one string argument: `prefix`. Return the target with each line prefixed by `prefix`. `["Hello\nWorld" columnPrefixedBy : "#␣"]` returns `"#␣Hello\n#␣World"`.

getter	Type	Meaning
`wrap`	string	Wraps the target to a width. This getter takes two int arguments: `width` and `shift`. The target is assumed to contain paragraphs separated by `\n`. Returns the target with each paragraph wrapped to `width`. In addition, each line of the paragraph except the first one is prefixed by `shift` spaces. `["Hello␣beautiful␣World.\nHow␣are ␣you" wrap : 6, 2]` returns `"Hello\n ␣␣beautiful\n␣␣World\nHow\n␣␣are\n you"`.
`subStringExists`	bool	This getter takes one argument, `subString`. It returns `true` if the sub-string `subString` is found in the target, `false otherwise`.
`replaceString`	string	This getter takes two argument, `find` and `replace`. It returns the target where each occurrence of `find` is replaced by `replace`.
`envVar`	string	Returns the value of the target environment variable. If it does not exists, `envVar` returns the empty string.
`envVarExists`	bool	Returns `true` if target environment variable exists, `false` otherwise.

1.5 The `bool` data type

A true literal bool can be written as `true` or `yes` and a false literal bool can be written as `false` or `no`.

1.5.1 `bool` operators

The `bool` data type supports the following operators:

Unary operator

Operator	Expression type	Meaning
`~`	`bool` ← `bool`	logical not

Binary operator

<table>
<tr><th>Operator</th><th>Expression type</th><th>Meaning</th></tr>
<tr><td><code>&</code></td><td><code>bool</code> ← <code>bool & bool</code></td><td>logical and</td></tr>
<tr><td><code>|</code></td><td><code>bool</code> ← <code>bool | bool</code></td><td>logical or</td></tr>
<tr><td><code>^</code></td><td><code>bool</code> ← <code>bool ^ bool</code></td><td>logical exclusive or</td></tr>
</table>

Comparison operators

For comparison operators, `false` is considered to be lower than `true`.

Operator	Expression type	Meaning
`!=`	`bool` ← `bool != bool`	Not equal
`==`	`bool` ← `bool == bool`	Equal
`>`	`bool` ← `bool > bool`	Greater than
`<`	`bool` ← `bool < bool`	Lower than
`>=`	`bool` ← `bool >= bool`	Greater or equal
`<=`	`bool` ← `bool <= bool`	Lower or equal

1.5.2 `bool` getters

getter	Type	Meaning
`trueOrFalse`	string	Returns a string representation, `"true"` or `"false"` of the bool expression.
`string`	string	Returns a string representation, `"true"` or `"false"` of the bool expression.
`yesOrNo`	string	Returns a string representation, `"yes"` or `"no"` of the bool expression.
`TRUEOrFALSE`	string	Returns a string representation, `"TRUE"` or `"FALSE"` of the bool expression.
`YESOrNO`	string	Returns a string representation, `"YES"` or `"NO"` of the bool expression.
`int`	int	Returns an int representation, `1` or `0` of the bool expression.

1.6 The `struct` data type

The struct data type allows to store a heterogeneous set of data in one variable. Struct members are accessed by using the `::` separator. If `A` is a struct, `A::B` refers to field `B` of `A`.

A literal struct is defined as follow:

```
@{ a: 1, b: 2, c: 3 }
```

This define a struct with fields a, b and c and respective values 1, 2 and 3.

1.6.1 `struct` operators

The `struct` data type supports the following operators:

Operator	Expression type	Meaning
`!=`	`bool` ← `struct != struct`	Not equal
`==`	`bool` ← `struct == struct`	Equal

Two structs are equal if:

- they have the same number of field
- they have the same field names
- they have the same field values

1.6.2 `struct` getter

getter	Type	Meaning
`map`	map	Returns a map representation.

1.7 The `list` data type

The list data type allows to store a list of data. list items are accessed by using `[<number>]` where `<number>` is the rank of the element starting at 0. If `A` is a list, `A[0]` refers to element 0 of `A`.

A literal list is defined as follow:

```
@( 1, 2, 3 )
```

This define a list of int with elements 1, 2 and 3.

An empty list can be initialized using the `emptylist` constant.

⚠ The **`emptylist`** constant is deprecated. Use a literal empty list, `@()`, instead.

1.7.1 `list` operators

The `list` data type supports the following operators:

Binary operators

Operator	Expression type	Meaning
`+`	`list` ← `list + any`	add any at the end of the list
`|`	`list` ← `list | list`	Concatenate lists

Comparison operators

Operator	Expression type	Meaning
`!=`	`bool` ← `list != list`	Not equal
`==`	`bool` ← `list == list`	Equal

Two structs are equal if:

- they have the same number of elements
- they have the same elements values

1.7.2 `list` getters

getter	Type	Meaning
`length`	int	Returns the number of elements in the list.
`first`	any	Returns the first element of the list.
`last`	any	Returns the last element of the list.
`mapBy`	map	`mapBy` takes a string argument which is the field (for a struct list item) or the key (for a map list item) used as key to store the element in the resulting map. It returns a map where each element is the element of the list with the key being the corresponding field/key.
`subListTo`	list	`subListTo` takes an int argument which is the stop `index` of the sublist. It returns a sublist which is a copy of target list ranging from 0 to the `index` included. If `aList` contains `@( 1, 2, 3, 4 )`, `[aList subListTo: 1]` returns `@( 1, 2 )`
`subListFrom`	list	`subListTo` takes an int argument which is the start `index` of the sublist. It returns a sublist which is a copy of target list ranging from `index` included to the end of the list. If `aList` contains `@( 1, 2, 3, 4 )`, `[aList subListFrom: 1]` returns `@( 2, 3, 4 )`
`subList`	list	`subList` takes 2 int arguments. The first one is the start `index` of the sublist. The second one is the `length` of the sublist. It returns a sublist which is a copy of target list ranging from `index` included to up to `length` items. If `aList` contains `@( 1, 2, 3, 4 )`, `[aList subList: 1, 5]` returns `@( 2, 3, 4 )`, `[aList subList: 2, 1]` returns `@( 3 )`

example of `mapBy`

The following code snippet:

```
let myList := @(
  @{
    age : 18,
    height : 180,
    name : "Arnold"
  },
  @{
    age : 22,
    height : 170,
    name : "Bob"
  },
  @{
    age : 29,
    height : 175,
    name : "John"
  }
)

let myMap := [myList mapBy : "name"]
display myMap
```

outputs:

```
myMap - map: @[
    "Arnold" :>
        struct: @{
            age :>
                integer: 18
            height :>
                integer: 180
            name :>
                string: "Arnold"
        }
    "Bob" :>
        struct: @{
            age :>
                integer: 22
            height :>
                integer: 170
            name :>
                string: "Bob"
        }
    "John" :>
        struct: @{
            age :>
                integer: 29
            height :>
                integer: 175
            name :>
```

```
                string: "John"
        }
]
```

1.7.3 `list` setters

getter	Type	Meaning
`insert`	list	`insert` takes 2 arguments. The first one is the `index` of the list where the data will be inserted. The second one is the `data` to insert. It inserts `data` before the item at `index`. If `aList` contains `@( 1, 2, 3, 4 )`, `[!aList insert: 1, "Hello"]` changes `aList` to `@( 1, "Hello", 2, 3, 4 )`

1.8 The `map` data type

The map data type allows to store an association of key and value. map members are accessed by using `[<key>]` where `<key>` is a *string*. If `A` is a map, `A["John"]` refers to an element of `A` having key `"John"`.

A literal map is defined as follow:

```
@[ "age" : 29, "height" : 175, "name" : "John" ]
```

An empty map can be initialized using the `emptymap` constant.

⚠ The **`emptymap`** constant is deprecated. Use a literal empty map, `@[]`, instead.

1.8.1 `map` operators

The `map` data type supports the following operators:

Operator	Expression type	Meaning
`!=`	`bool ← map != map`	Not equal
`==`	`bool ← map == map`	Equal

Two maps are equal if:

- they have the same number of items
- they have the same item keys
- they have the same item values

1.8.2 `map` getters

getter	Type	Meaning
`length`	int	Returns the number of elements in the map.
`list`	any	Returns a list representation of the map. Elements of the list are in the alphanumerical order of the keys of the map.

2 GTL instructions

2.1 The %. . . % instruction

The %. . . % is the literal template string instruction. Every characters appearing between `%` are accumulated in the output string of the template. GTL starts by assuming a `%` exists just before the first character of the file. So if the first character of the file is not a `%` the first instruction is a %. . . % instruction up to the first `%` in the file.

2.2 The *let* instruction

let is the data assignment instruction. The general form is:

```
let var := expression
```

If the variable does not exists, it is created. The variable is set to *expression*
If the `:= expression` is omitted, the variable is created and is unconstructed:

```
let var
```

As in the C language, GTL has assignment operators. For instance to increment an `int` variable, one can write:

```
let var += 1
```

The following table gives the available assignment operators and their meaning.

Assign.	int	float	string	bool	struct	list	map	uncons
+=	+	+	concat	NA	NA	add	NA	NA
-=	-	-	NA	NA	NA	NA	NA	NA
*=	*	*	NA	NA	NA	NA	NA	NA
/=	/	/	NA	NA	NA	NA	NA	NA
mod=	mod	NA	NA	NA	NA	NA	NA	NA
<<=	<<	NA	NA	NA	NA	NA	NA	NA
>>=	>>	NA	NA	NA	NA	NA	NA	NA
&=	bitwise &	NA	NA	logical &	NA	NA	NA	NA
|=	bitwise |	NA	NA	logical |	NA	NA	NA	NA
^=	bitwise ^	NA	NA	logical ^	NA	NA	NA	NA

The scope of a variable depends on the location where the variable is assigned the first time. For instance, in the following code:

```
let a := 1
foreach task in TASKS do
  let b := 2
  let a += 1
end foreach
println a
println b
```

Because a is assigned outside the **foreach** loop, it is both accessible within the foreach loop and accessible after the **foreach** loop. So it contains the number of items in TASKS + 1 after the **foreach**. Because b is assigned inside the **foreach** loop, it does not exist after the loop anymore and **println** b will trigger and error.

2.3 The *unlet* instruction

The *unlet* instruction removes a variable, a struct field, a map item or a list item. The variable / struct field / map item / list item ceases to exist. If the variable / struct field / map item / list does not exist, *unlet* fails silently. Here are some examples. The following program:

```
let a := 0

if exists a then
  println "'a' found"
else
  println "'a' not found"
end if

unlet a

if exists a then
  println "'a' found"
else
  println "'a' not found"
end if
```

outputs:

```
'a' found
'a' not found
```

Here *unlet* is used to remove a field from a struct:

```
let myStruct := @{ a: 1, b: 2, c: 3 }
unlet myStruct::a
display myStruct
```

and produces the following output

```
myStruct - struct: @{
    b :>
        integer: 2
    c :>
        integer: 3
}
```

Here we use *unlet* to remove an item from a list:

```
let myList := @( 1, 2, 3, 4 )
unlet myList[2]
display myList
```

The execution produces the following output:

```
myList - list: @(
    0 :>
        integer: 1
    1 :>
        integer: 2
    2 :>
        integer: 4
)
```

And here to remove an item from a map

```
let myMap := @[ "a": @( 1, 2) , "b": @( 3, 4) ]
unlet myMap["b"]
display myMap
```

The execution produces the following output:

```
myMap - map: @[
    "a" :>
        list: @(
            0 :>
                integer: 1
            1 :>
                integer: 2
        )
]
```

2.4 The *!* instruction

The *!* instruction emits an expression in the output template string. The syntax is

```
! expression
```

For instance the following program:

```
loop i from 1 to 10 do
  !" " !i
end loop
%
%
```

produces the following output string:

```
 1 2 3 4 5 6 7 8 9 10
```

2.5 The *if* instruction

Conditional execution. The forms are:

```
if expression then
  instruction_list
end if

if expression then
  instruction_list
else
  instruction_list
end if

if expression then
  instruction_list
elsif expression then
  instruction_list
end if

if expression then
  instruction_list
elsif expression then
  instruction_list
else
  instruction_list
end if
```

The *expression* must be boolean. In the following example, the blue text (within the %) is produced only if the `USECOM` boolean variable is true:

```
if USECOM then %
#include "tpl_com.h" %
end if
```

2.6 The *foreach* instruction

This instruction iterates on the elements of a collection, a list or a map. The simplest form is the following one:

```
foreach var in expression do
  instruction_list
end foreach
```

Here var takes the value of each of the elements of the collection. If the collection is a list, the elements are iterated in the order of the list. If the collection is a map, the element are iterated in the alphanumerical order of the keys. In both cases, a variable named `INDEX` which contains the current iteration number is available inside the loop. `INDEX` ranges from 0 to the number of elements in the list minus 1. If the collection is a map, a second variable, `KEY`, which contains the key associated to the value of the current item, is available.

In the following example, for each element in the `ALARMS` list, the text between the **`do`** and the **`end foreach`** is produced with the `NAME` attribute of the current element of the `ALARMS` list inserted at the specified location.

```
foreach alr in ALARMS do
%
/* Alarm % !alr::NAME % identifier */
#define % !alr::NAME %_id % !INDEX %
CONST(AlarmType, AUTOMATIC) % !alr::NAME % = % !NAME %_id;
%
end foreach
```

A more general form of the **foreach** instruction is:

```
foreach key,var (index) in expression
  before
    instruction_list
  do
    instruction_list
  between
    instruction_list
  after
    instruction_list
end foreach
```

`key` may be used only when iterating on a map and allows to give a custom name to the default `KEY` variable. `(`*`index`*`)` may be used both for a list or a map and allows to give a custom name to the default `INDEX` variable.

If the collection is not empty, the **before** section is executed once before the first execution of the **do** section. If the collection contains at least two elements, the **between** section is executed between the execution of the **do** section. If the list is not empty, the **after** section is executed once after the last execution of the **do** section.

The following example illustrates the general form. Here a table of pointers to alarm descriptors is generated:

```
#
# Initialize ALARMS with a list of 2 structs with a NAME field.
#
let ALARMS := @( @{ NAME: "alr1"}, @{ NAME: "alr1"} )

foreach alr in ALARMS
  before %
tpl_time_obj *tpl_alarm_table[ALARM_COUNT] = {
%
  do %  &% !alr::NAME %_alarm_desc%
  between %,
%
  after %
};
%
end foreach
```

It produces the following output:

```
tpl_time_obj *tpl_alarm_table[ALARM_COUNT] = {
```

```
  &alr1_alarm_desc,
  &alr1_alarm_desc
};
```

2.7 The *for* instruction

The *for* instruction iterates along a literal list of elements.

```
for var in expression, ... , expression do
  ...
end for
```

At each iteration, *var* gets the value of the current *expression*. As in the **foreach** instruction, `INDEX` is generated and ranges from 0 to the number of elements in the list minus 1.

The **for** instruction is deprecated. Use **foreach** with a literal list instead.

2.8 The *loop* instruction

The *loop* instruction iterate over a range of integers. Its simplest form is:

```
loop var from expression_start to expression_end do
  ...
end loop
```

Both *expression_start* and *expression_end* must be integer expressions. By default `var` is incremented by one from *expression_start*, inclusive, to *expression_end*, inclusive.

Like in the foreach instruction, **before**, **between** and **after** sections may be used. Moreover, **down** may be used to decrement *var* by one. **up** is a syntactic sugar which is here for symmetry purpose and may be omitted. **step** allows to increment or decrement by *increment*. If **step** is omitted, **step** `1` is assumed.

```
loop var from expression <up|down> to expression <step increment>
  before ...
  do ...
  between ...
  after ...
end loop
```

For instance, in the following loop, `a` goes from 0 to 10 with an increment of 2:

```
loop a from 0 to 10 step 2 do
  display a
end loop
```

and produces the following output:

```
integer: 0
integer: 2
integer: 4
```

```
integer: 6
integer: 8
integer: 10
```

In the following loop, a goes from 25 to 20 with a decrement of 1:

```
loop a from 25 down to 20 do
  display a
end loop
```

and produces the following output:

```
integer: 25
integer: 24
integer: 23
integer: 22
integer: 21
integer: 20
```

Because the step can be a negative integer number, this output may be produced by the following program too:

```
loop a from 25 to 20 step -1 do
  display a
end loop
```

⚠ Despite the use of big integers the number of iteration is limited to $2^{32} - 1$

2.9 The *repeat* instruction

The *repeat* instruction combine the C `while (...) { ... }` and `do { ... } while (...);` in one instruction. The general form is:

```
repeat <(limit_expression)>
  instruction_list_1
while condition do
  instruction_list_2
end repeat
```

`limit_expression` is an optional expression to bound the number of iterations. If the number of iterations exceeds `limit`, a runtime error is emitted. If `limit_expression` is omitted, its default value is $2^{32} - 1$.

The semantics of this instruction is shown at figure 1

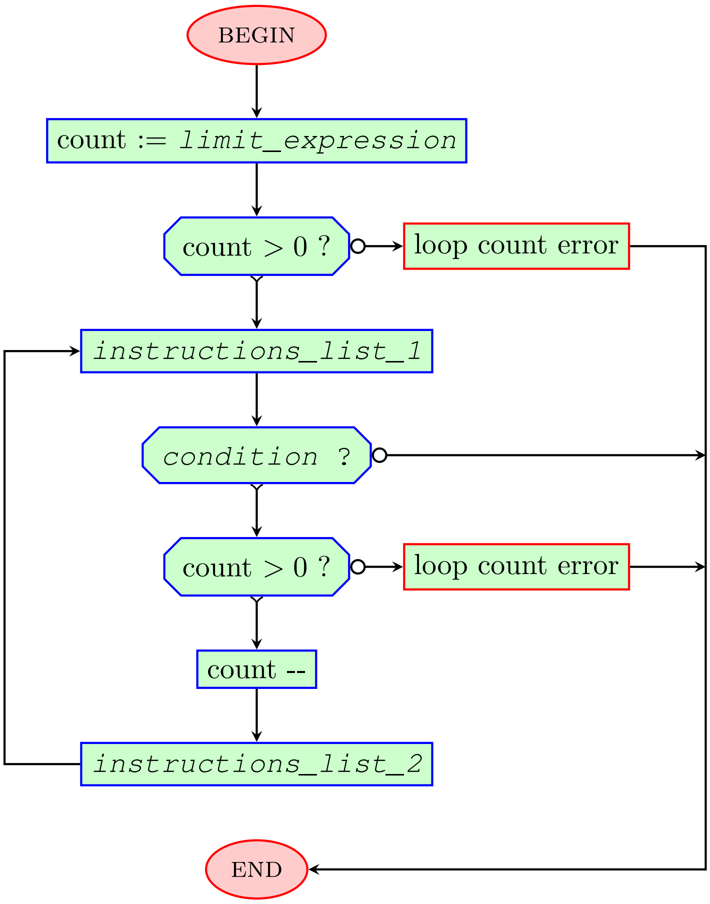

Figure 1: semantics of the *repeat* instruction