

GTL Template language

Jean-Luc Béchenec

August 2, 2016

Contents

1	Data types	3
1.1	getters applicable to any data type	3
1.2	int data type	4
1.2.1	int operators	4
1.2.2	int getters	5
1.2.3	int setters	6
1.3	The float data type	6
1.3.1	float operators	6
1.3.2	float getters	7
1.4	The string data type	7
1.4.1	string operators	8
1.4.2	string getters	8
1.5	The bool data type	11
1.5.1	bool operators	11
1.5.2	bool getters	12
1.6	The struct data type	12
1.6.1	struct operators	13
1.6.2	struct getter	13
1.7	The list data type	13
1.7.1	list operators	13
1.7.2	list getters	14
1.8	The map data type	15
1.8.1	map operators	15
2	GTL instructions	16
2.1	The <code>%...%</code> instruction	16
2.2	The <code>let</code> instruction	16
2.3	The <code>unlet</code> instruction	17
2.4	The <code>if</code> instruction	18
2.5	The <code>foreach</code> instruction	18
2.6	The <code>for</code> instruction	20

2.7	The <i>loop</i> instruction	20
2.8	The <i>repeat</i> instruction	21

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
+	<code>int ← +int</code>	Plus operator. No effect
-	<code>int ← -int</code>	Minus operator. Negation
~	<code>int ← ~int</code>	Not operator. Complementation by 1

Binary arithmetic operators

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

Binary bitwise operators

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

Comparison operators

Operator	Expression type	Meaning
!=	<code>bool ← int != int</code>	Not equal
==	<code>bool ← int == int</code>	Equal
>	<code>bool ← int > int</code>	Greater than
<	<code>bool ← int < int</code>	Lower than
>=	<code>bool ← int >= int</code>	Greater or equal
<=	<code>bool ← int <= int</code>	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
<code>fitsSignedInLong</code>	<code>bool</code>	Returns <code>true</code> if the expression fits in a signed 32 bits long, <code>false</code> otherwise.
<code>fitsUnsignedInLongLong</code>	<code>bool</code>	Returns <code>true</code> if the expression fits in an unsigned 64 bits long long, <code>false</code> otherwise.
<code>fitsSignedInLongLong</code>	<code>bool</code>	Returns <code>true</code> if the expression fits in a signed 64 bits long long, <code>false</code> otherwise.
<code>abs</code>	<code>int</code>	Returns the absolute value of the expression.
<code>bitAtIndex</code>	<code>bool</code>	This getter takes one argument: <code>index</code> . It returns <code>true</code> if the bit at index <code>index</code> is set and <code>false</code> otherwise. <code>index 0</code> corresponds to the lowest significant bit. [<code>1 bitAtIndex: 0</code>] returns <code>true</code>

1.2.3 int setters

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

1.3 The float data type

The `float` data type is the standard IEEE754 64 bits floating point number.

1.3.1 float operators

The `float` data type supports the following operators:

Unary operators

Operator	Expression type	Meaning
<code>+</code>	<code>float ← +float</code>	Plus operator. No effect
<code>-</code>	<code>float ← -float</code>	Minus operator. Negation

Binary arithmetic operators

Operator	Expression type	Meaning
<code>+</code>	<code>float ← float + float</code>	Addition

Operator	Expression type	Meaning
-	float \leftarrow float - float	Substraction
*	float \leftarrow float * float	Multiplication
/	float \leftarrow float / float	Division

Comparison operators

Operator	Expression type	Meaning
!=	bool \leftarrow float != float	Not equal
==	bool \leftarrow float == float	Equal
>	bool \leftarrow float > float	Greater than
<	bool \leftarrow float < float	Lower than
>=	bool \leftarrow float >= float	Greater or equal
<=	bool \leftarrow 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
<code>\t</code>	horizontal tab
<code>\v</code>	vertical tab
<code>\\</code>	backslash
<code>\0</code>	null character
<code>\unnnn</code>	unicode character with code <i>nnnn</i> in hexadecimal
<code>\Unnnnnnnn</code>	unicode character with code <i>nnnnnnnn</i> in hexadecimal

1.4.1 string operators

The string data type supports the following operators:

Binary operator

Operator	Expression type	Meaning
<code>+</code>	<code>string ← string + string</code>	Concatenation

Comparison operators

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

1.4.2 string getters

getter	Type	Meaning
<code>HTMLRepresentation</code>	string	Returns a representation of the string suitable for an HTML encoded representation. <code>'&'</code> is encoded by <code>&amp;</code> ; <code>'"</code> by <code>&quot;</code> ; <code>'<'</code> by <code>&lt;</code> ; and <code>'>'</code> by <code>&gt;</code> .

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. ["HelloWorld!"] 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. ["HelloWorld!"] 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. ["HelloWorld!"] 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\n How\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

Operator	Expression type	Meaning
&	bool \leftarrow bool & bool	logical and
	bool \leftarrow bool bool	logical or
^	bool \leftarrow bool ^ bool	logical exclusive or

Comparison operators

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

Operator	Expression type	Meaning
!=	bool \leftarrow bool != bool	Not equal
==	bool \leftarrow bool == bool	Equal
>	bool \leftarrow bool > bool	Greater than
<	bool \leftarrow bool < bool	Lower than
>=	bool \leftarrow bool >= bool	Greater or equal
<=	bool \leftarrow 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
<code>!=</code>	<code>bool ← struct != struct</code>	Not equal
<code>==</code>	<code>bool ← struct == struct</code>	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
<code>map</code>	<code>map</code>	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
<code>+</code>	<code>list ← list + any</code>	add any at the end of the list
<code> </code>	<code>list ← list list</code>	Concatenate lists

Comparison operators

Operator	Expression type	Meaning
<code>!=</code>	<code>bool ← list != list</code>	Not equal
<code>==</code>	<code>bool ← list == list</code>	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
<code>length</code>	<code>int</code>	Returns the number of elements in the list.
<code>first</code>	<code>any</code>	Returns the first element of the list.
<code>last</code>	<code>any</code>	Returns the last element of the list.
<code>mapBy</code>	<code>map</code>	<code>mapBy</code> takes on 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.

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"
    }
]
```

1.8 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
<code>!=</code>	<code>bool ← map != map</code>	Not equal
<code>==</code>	<code>bool ← map == map</code>	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

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 operator:

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

2.4 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
```

2.5 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
};
```

2.6 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.

2.7 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 $2^{32} - 1$

2.8 The *repeat* instruction

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

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