GTL Template language

Jean-Luc Béchennec

July 29, 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"`.

getter	Type	Meaning
`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.
`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.

getter	Type	Meaning
`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
`-`	`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
`\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 `&quot;` , '`<`' by `<` and '`>`' by `>` .
`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

getter	Type	Meaning
`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"`.
`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"`.

getter	Type	Meaning
`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"`.
`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\nyou"`.
`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 ← bool & bool`	logical and
`|`	`bool ← bool | bool`	logical or
`^`	`bool ← bool ^ bool`	logical exclusive or

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 in one variable. list members 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.

1.7.1 `list` operators

The `list` data type supports the following operators:

Binary operators

Operator	Expression type	Meaning
`+`	`list ← list + any`	add `any` add 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 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:

```
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"
        }
]
```