The GTL Template language

version 3

Jean-Luc Béchennec

November 2, 2020

Contents

Contents **1**

1 About this document **3**

2 Data types **5**
2.1 Operators priority . . . 6
2.2 Special operators . . . 6
2.3 Getters applicable to any data type . . . 9
2.4 Setters applicable to any data type . . . 10
2.5 The `int` data type . . . 10
2.6 The `char` data type . . . 16
2.7 The `enum` data type . . . 18
2.8 The `float` data type . . . 18
2.9 The `string` data type . . . 20
2.10 The `bool` data type . . . 27
2.11 The `struct` data type . . . 28
2.12 The `list` data type . . . 29
2.13 The `map` data type . . . 33
2.14 The `set` data type . . . 34
2.15 The `type` data type . . . 37

3 GTL instructions **39**
3.1 The `%...%` instruction . . . 39
3.2 The `let` instruction . . . 39
3.3 The `unlet` instruction . . . 40
3.4 The `!` instruction . . . 42
3.5 The `?` instruction . . . 42
3.6 The `tab` instruction . . . 42
3.7 The `sort` instruction . . . 42
3.8 The `if` instruction . . . 44
3.9 The `foreach` instruction . . . 45
3.10 The `for` instruction . . . 46
3.11 The `loop` instruction . . . 47
3.12 The `repeat` instruction . . . 48
3.13 The `write` instruction . . . 49
3.14 The `template` instruction . . . 49
3.15 The `input` instruction . . . 51

3.16 The `error` and `warning` instructions . . . 52
3.17 The `print` and `println` instructions . . . 52
3.18 The `display` instruction . . . 53
3.19 The `variables` instruction . . . 54
3.20 The `seed` instruction . . . 55

4 GTL modules **57**
4.1 Importing a module . . . 57
4.2 Writing a module . . . 57
4.3 Getter definition . . . 59
4.4 Setter definition . . . 61

5 The GTL debugger **63**
5.1 The help command . . . 63

6 Using GTL in a GALGAS project **65**
6.1 Needed files . . . 65
6.2 Invoking a template . . . 65

1

About this document

This document presents the syntax and the semantics of GTL 3, a template language used in Goil, the OIL compiler of Trampoline[1], to generate code from the OIL or arXML description of an OSEK/AUTOSAR application.

GTL is written in GALGAS 3[2], a lexical and syntactic analyzer, which is also a powerful domain specific language to write compilers and interpreters. GALGAS 3 is developed by Pierre Molinaro from the Real-Time Systems Group of IRCCyN, a french laboratory from a joint of the University of Nantes, École Centrale de Nantes, École des Mines de Nantes and CNRS. The first version of the GTL interpreter was written by Pierre Molinaro too. As needs have increased, more features was added and this led to a major rewrite of the GTL interpreter; In fact no original code made its way in GTL 3.

Convention

Code examples follow the following conventions:

- GTL example code snippets are presented in light blue boxes. In these examples, *Italic* writing are used for generic syntactic items. For instance, `expression` means any expression like `a + b` or `"a string"`. Boldface words are reserved words of GTL, like **`foreach`** or built-in functions, getters or setters like **`setDescription`**. Pieces of code delimited by `<` and `>` are optional.
- Standard output of examples are presented in light yellow boxes.
- Template string output of examples are presented in light green boxes.

[1] An OSEK and AUTOSAR 4.2 compliant RTOS, check https://github.com/TrampolineRTOS/trampoline.

[2] GALGAS 3 is free software distributed under the GPL license and can be found at http://galgas.rts-software.org

2

Data types

GTL supports the following data types:

type	Description
int	arbitrary precision integer numbers. The GMP library is used
char	unicode chars
float	64 bits floating point numbers
bool	standard boolean
enum	enumerated type
string	unicode strings
struct	structured data
list	lists of data, may be accessed as a table
map	map (aka dictionary) of data
set	set of strings
type	the type of a data
unconstructed	an unconstructed variable

Data embed a location, that is a file name, a line and a column. Each time a variable is set, the corresponding location is set too. This allow to report errors efficiently. See 2.4 and 3.16.

Data embed a description string too. This allow to comment on the content of the data. See 2.3 and 2.4.

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 target but do not change it. They are used to do a computation with the target as input or to convert it into another type. Setters may only target a variable. Setters change the content of the target and do not return anything. Getters and setters may have arguments. Syntax for getters without argument is as follow:

```
[expression aGetter]
```

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

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

Syntax for setters without arguments is as follow:

```
[!variable aSetter]
```

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

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

2.1 Operators priority

The following table gives all the operators available in GTL an their priority. Semantics of the operators is given for each data type. Operators of same priority are evaluated from left to right.

Priority	Operators
0	| ^
1	&
2	== != < > <= >=
3	« » + . -
4	* / mod
5	not ~ - + typeof mapof listof exists

2.2 Special operators

The `exists` operator

The `exists` operator tests the existence of a variable, a struct field, an item of a map or a list and returns a bool, `true` if the entity exists, `false` otherwise.

```
exists var
```

Example

```
let c := 3
if exists c then println "c exists" else println "c does not exist" end if
unlet c # delete c
if exists c then println "c exists" else println "c does not exist" end if
```

```
c exists
c does not exist
```

The `exists ... default (...)` operator

The `exists ... default (...)` operator tests the existence of a variable, a struct field, an item of a map or a list.

```
exists var default ( expression )
```

If the entity exists it is returned. Otherwise the evaluation of the default expression is returned.

Example

```
let aList := @( 1, 2, 3 )
let aSecondList := exists aList default ( @() )
display aSecondList
unlet aList # delete aList
let aSecondList := exists aList default ( @() )
display aSecondList
```

```
aSecondList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 9:7
    list: @(
        0 :>
            integer: 1
        1 :>
            integer: 2
        2 :>
            integer: 3
    )
aSecondList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 12:7
    list: @(
    )
```

The `typeof` operator

The `typeof` operator takes a variable as argument an return its type.

```
typeof var
```

⚠ The `typeof` is deprecated. Use the `type` getter instead. See 2.3.

The `mapof` operator

The `mapof` may apply on `struct` or on `list`.

With a `struct`

The `mapof` operator converts a `struct` to a `map`.

```
mapof expression end
```

⚠ `mapof` is deprecated, use the map getter on `struct` instead. See 2.11.

Example

```
let aStruct := @{ a: 1, b: 2, c: 3 }
display aStruct
let aMap := mapof aStruct end
display aMap
```

```
aStruct from file '/Users/jlb/Develop/GTL/examples/mapofTest.gtl', line 3:7
    struct: @{
        a :>
            integer: 1
```

```
        b :>
            integer: 2
        c :>
            integer: 3
    }
aMap from file '/Users/jlb/Develop/GTL/examples/mapofTest.gtl', line 5:7
    map: @[
        "a" :>
            integer: 1
        "b" :>
            integer: 2
        "c" :>
            integer: 3
    ]
```

With a `list`

The **`mapof`** operator converts a `list` to a map according to a `string` field of each item of the `list`. So the list should be a `list` of `struct`.

```
mapof var by identifier
```

⚠ **`mapof ... by`** is deprecated, use the **`mapBy`** getter on `list` instead. See 2.12.

Example

```
let aList := @(
  @{ age : 18, height : 180, name : "Arnold" },
  @{ age : 22, height : 170, name : "Bob"    },
  @{ age : 29, height : 175, name : "John"   }
)
let aMap := mapof aList by name
display aMap
```

```
aMap from file '/Users/jlb/Develop/GTL/examples/mapofTest.gtl', line 13:7
    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"
            }
    ]
```

The `listof` operator

The **`listof`** operator apply to a `map` variable. It returns a `list` representation of the `map`. The elements of the `list` are sorted in the alphanumerical order of the `map` keys.

```
listof var end
```

⚠ **`listof ... end`** is deprecated, use the **`list`** getter on `map` instead. See 2.13.

2.3 Getters applicable to any data type

The `type` getter

The **`type`** getter returns the type of the expression. See the section 2.15.

Example

```
let a := 1
let typeOfA := [a type]
display typeOfA
```

```
typeOfA from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 4:7
    type: int
```

The `isANumber` getter

`isANumber` returns **`true`** if the expression is a number: `int` or `float`, **`false`** otherwise. This getter is useful to test the type of an argument in a function, a getter or a setter.

Example

```
let b := 0
let a := 1
if [a isANumber] then
  let b := a # if a is a number, it is copied in b
else
  let b := 0 # otherwise b is set to 0
end if
display b
```

```
b from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 13:7
    integer: 1
```

The `description` getter

`description` returns the string describing the data, if available, or an empty string otherwise. If the data is coming from an OIL source file, it corresponds to the description field (see section 2.3.11 of *System Generation – OIL: OSEK Implementation Language – Version 2.5*).

2.4 Setters applicable to any data type

The `setDescription` setter

`setDescription` takes one string argument `desc`. It sets the string describing the data to `desc`.

The `touch` setter

`touch` sets the location of modification of the data to the current one.

2.5 The `int` data type

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

The `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

Operator	Expression type	Meaning
»	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

The int getters

The string getter

`string` returns a string decimal representation of the `int` expression.

Example

```
let b := [42 string]
display b
```

```
b from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 18:7
    string: "42"
```

The hexString getter

`hexString` returns a hexadecimal string representation of the int expression prefixed by `"0x"`. If the int expression is negative a `'-'` is inserted before.

Example

```
let a := [42 hexString]
display a
let b := [-20 hexString]
display b
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 21:7
    string: "0x2A"
b from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 23:7
    string: "-0x14"
```

the xString getter

`xString` returns a hexadecimal string representation of the int expression. If the expression is negative a `'-'` is inserted before.

Example

```
let a := [42 xString]
display a
let b := [-20 xString]
display b
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 26:7
    string: "2A"
b from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 28:7
    string: "-14"
```

The numberOfBytes getter

`numberOfBytes` returns an int, the number of bytes needed to store an unsigned expression.

Example

```
println [255 numberOfBytes]
println [256 numberOfBytes]
```

```
1
2
```

The signedNumberOfBytes getter

`signedNumberOfBytes` returns an int, the number of bytes needed to store a signed expression.

Example

```
println [127 signedNumberOfBytes]
println [128 signedNumberOfBytes]
```

```
1
2
```

The numberOfBits getter

`numberOfBits` returns an int, the number of bits needed to store an unsigned expression.

Example

```
println [63 numberOfBits]
println [64 numberOfBits]
```

```
6
7
```

The signedNumberOfBits getter

`signedNumberOfBits` returns an int, the number of bits needed to store a signed expression.

Example

```
println [63 signedNumberOfBits]
println [64 signedNumberOfBits]
```

```
7
8
```

The sign getter

`sign` returns an int, `-1` if the expression is strictly negative, `0` if it is null and `+1` if the expression is strictly positive.

The fitsUnsignedInByte getter

`fitsUnsignedInByte` returns a bool, `true` if the expression fits in an unsigned byte, `false` otherwise.

The fitsSignedInByte getter

`fitsSignedInByte` returns a bool, `true` if the expression fits in a signed byte, `false` otherwise.

The fitsUnsignedInWord getter

`fitsUnsignedInWord` returns a bool, `true` if the expression fits in an unsigned 16 bits word, `false` otherwise.

The fitsSignedInWord getter

`fitsSignedInWord` returns a bool, `true` if the expression fits in a signed 16 bits word, `false` otherwise.

The fitsUnsignedInLong getter

`fitsUnsignedInLong` returns a bool, `true` if the expression fits in an unsigned 32 bits long, `false` otherwise.

The fitsSignedInLong getter

`fitsSignedInLong` returns a bool, `true` if the expression fits in a signed 32 bits long, `false` otherwise.

The fitsUnsignedInLongLong getter

`fitsUnsignedInLongLong` returns a bool, `true` if the expression fits in an unsigned 64 bits long long, `false` otherwise.

The `fitsSignedInLongLong` getter

`fitsSignedInLongLong` returns a bool, `true` if the expression fits in a signed 64 bits long long, `false` otherwise.

The `abs` getter

`abs` returns an int, the absolute value of the expression.

The `bitAtIndex` getter

`bitAtIndex` takes one int argument: `index`. It returns `true` if the bit at index `index` is set and `false` otherwise. `index` 0 corresponds to the lowest significant bit.

Example

```
let a := [1 bitAtIndex: 0]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 63:7
    boolean: true
```

The `int` setters

The `setBitAtIndex` setter

`setBitAtIndex` 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.

Example

```
let a := 0
[!a setBitAtIndex: true, 0]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 64:7
    integer: 1
```

The `complementBitAtIndex` setter

`complementBitAtIndex` takes one int argument, `index`, which is the index of the bit to complement.

Example

```
let a := 1
[!a complementBitAtIndex: true, 1]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 68:7
    integer: 3
```

The int functions

The following built-in functions return an int.

The function majorVersion()

`majorVersion()` takes no argument and returns the major version of the compiler in where GTL is embedded. For instance, if the version string is `"3.2.1"`, it returns `3`.

The function minorVersion()

`minorVersion()` takes no argument and returns the minor version of the compiler in where GTL is embedded. For instance, if the version string is `"3.2.1"`, it returns `2`.

The function revision()

`revision()` takes no argument and returns the revision of the compiler in where GTL is embedded. For instance, if the version string is `"3.2.1"`, it returns `1`.

The function max8bitsUnsignedInt()

`max8bitsUnsignedInt()` takes no argument and returns the maximum unsigned number that fits in 8 bits.

The function max8bitsSignedInt()

`max8bitsSignedInt()` takes no argument and returns the maximum signed number that fits in 8 bits.

The function min8bitsSignedInt()

`min8bitsSignedInt()` takes no argument and returns the minimum signed number that fits in 8 bits.

The function max16bitsUnsignedInt()

`max16bitsUnsignedInt()` takes no argument and returns the maximum unsigned number that fits in 16 bits.

The function max16bitsSignedInt()

`max16bitsSignedInt()` takes no argument and returns the maximum signed number that fits in 16 bits.

The function min16bitsSignedInt()

`min16bitsSignedInt()` takes no argument and returns the minimum signed number that fits in 16 bits.

The function max32bitsUnsignedInt()

`max32bitsUnsignedInt()` takes no argument and returns the maximum unsigned number that fits in 32 bits.

The function `max32bitsSignedInt()`

`max32bitsSignedInt()` takes no argument and returns the maximum signed number that fits in 32 bits.

The function `min32bitsSignedInt()`

`min32bitsSignedInt()` takes no argument and returns the minimum signed number that fits in 32 bits.

The function `max64bitsUnsignedInt()`

`max64bitsUnsignedInt()` takes no argument and returns the maximum unsigned number that fits in 64 bits.

The function `max64bitsSignedInt()`

`max64bitsSignedInt()` takes no argument and returns the maximum signed number that fits in 64 bits.

The function `min64bitsSignedInt()`

`min64bitsSignedInt()` takes no argument and returns the minimum signed number that fits in 64 bits.

The function `random(min, max)`

`random(min, max)` returns a random number between `min` included and `max` excluded. `min` and `max` have to fit in an unsigned 32 bits integer.

2.6 The `char` data type

The char data type supports unicode characters. A literal char is delimited by a pair of `'`:

```
let a := 'A'
```

Literal chars support escaped 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

The char operators

The char data type supports the following operators:

Comparison operators

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

The char getters

The string getter

`string` returns a string representation of the char expression. `['c' string]` returns string `"c"`.

The isAlnum getter

`isAlnum` returns a bool, `true` if the char expression is an ASCII alphanumeric character: between `'A'` and `'Z'` or between `'a'` and `'z'` or between `'0'` and `'9'`, `false` otherwise.

The isAlpha getter

`isAlpha` returns a bool, `true` if the char expression is an ASCII letter: between `'A'` and `'Z'` or between `'a'` and `'z'`, `false` otherwise.

The isDigit getter

`isDigit` returns abool, `true` if the char expression is an ASCII digit: between `'0'` and `'9'`, `false` otherwise.

The isCntrl getter

`isCntrl` returns a bool, `true` if the char expression is an ASCII control character, i.e. strictly before the *SPACE* character, `false` otherwise.

The isLower getter

`isLower` returns a bool, `true` if the char expression is an ASCII lower case letter: between `'a'` and `'z'`, `false` otherwise.

The isUpper getter

`isUpper` returns a bool, `true` if the char expression is an ASCII upper case letter: between `'A'` and `'Z'`, `false` otherwise.

The `isXDigit` getter

`isXDigit` returns a bool, `true` if the char expression is an ASCII hexadecimal digit: between `'A'` and `'F'` or between `'a'` and `'f'` or between `'0'` and `'9'`, `false` otherwise.

2.7 The `enum` data type

The enum data type allow to store identifiers in variables. A literal enum begins with a `'$'` followed with a name composed of ASCII letters, numbers and `'_'`.

Example

```
let a := $auto
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/enumTest.gtl', line 4:7
    enum: auto
```

The `enum` operators

Comparison operators

Operator	Expression type	Meaning
!=	bool ← enum != enum	Not equal
==	bool ← enum == enum	Equal

The `enum` getters

The `string` getter

`string` returns a string representation of the enum expression. `[@auto string]` returns string `"auto"`.

2.8 The `float` data type

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

The `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

The float getters

The string getter

`string` returns a string representation of the float expression. `[4.2 string]` returns string `"4.2"`.

The cos getter

`cos` returns the cosine of a float expression expressed in radian.

The sin getter

`sin` returns the sine of a float expression expressed in radian.

The tan getter

`tan` returns the tangent of a float expression expressed in radian.

The cosDegree getter

`cosDegree` returns the cosine of a float expression expressed in degree.

The sinDegree getter

`sinDegree` returns the sine of a float expression expressed in degree.

The tanDegree getter

`tanDegree` returns the tangent of a float expression expressed in degree.

The exp getter

`exp` returns the exponentiation of a float expression.

The `logn` getter

`logn` returns the natural logarithm of a float expression.

The `log2` getter

`log2` returns the logarithm base 2 of a float expression.

The `log10` getter

`log10` returns the logarithm base 10 of a float expression.

The `sqrt` getter

`sqrt` returns the square root of a float expression.

The `power` getter

`power` takes one `float` argument, `p`. It returns the expression raised to the power of `p`.

The `float` function

The following function returns a `float`

The `pi()` function

`pi()` returns an approximation of the π constant value (3.14159265358979323846264338327950288).

2.9 The `string` data type

The `string` data type supports unicode. A literal string is delimited by a pair of ".

```
let a := "A literal string"
```

Literal strings support escaped 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

The 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

The string getters

The unsigned getter

`unsigned` interprets the content of the **`string`** as an unsigned integer and returns it.

The charAtIndex getter

`charAtIndex` takes one int argument, the `index`. It returns a char, the character at index `index`. The first char is at index 0, the last char is at index length of the string minus 1. If `index` is greater or equal than the length of the target, a run time error occurs.

Example

```
let a := ["Hello" charAtIndex: 1]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 74:7
    char: e
```

The indexOfChar getter

`indexOfChar` takes one char argument, the `character` to look up. It returns an int, the index of the first occurrence of `character` in the target. If `character` is not found in the target, `-1` is returned.

Example

```
let a := ["Hello" indexOfChar: 'l']
let b := ["Hello" indexOfChar: 'z']
display a
display b
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 78:7
    integer: 2
b from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 79:7
    integer: -1
```

The `indexOfCharInRange` getter

`indexOfCharInRange` takes two char argument, `minChar` and `maxChar` which define the character range to look up. It returns an int, the index of the first occurrence of the character being in range (bounds included) in the target. If `character` is not found in the target, `-1` is returned.

Example

```
let a := ["Hello" indexOfCharInRange: 'a', 'e']
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 82:7
    integer: 1
```

The `containsChar` getter

`containsChar` takes one char argument, the `character` to look up. It returns a bool, `true` is the target contains the `character`, `false` otherwise.

The `containsCharInRange` getter

`containsCharInRange` takes two char argument, `minChar` and `maxChar` which define the character range to look up. It returns a bool, `true` is the target contains a character within the range (bounds included), `false` otherwise.

The `HTMLRepresentation` getter

`HTMLRepresentation` returns a representation of the string suitable for an HTML encoded representation. `'&'` is encoded by `&` , `'"'` by `&quot;` , `'<'` by `<` and `'>'` by `>`.

The `identifierRepresentation` getter

`identifierRepresentation` 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`.

The `fileExists` getter

`fileExists` returns a bool, `true` if a file exists at the target path, `false` otherwise.

The `files` getter

`files` returns a set, each element being a regular file in the directory specified by the string..

The length getter

`length` returns and int, the number of characters in the string.

The lowercaseString getter

`lowercaseString` returns the lowercased representation of the string.

The capitalized getter

If the string is empty, `capitalized` returns the empty string; otherwise, it returns the string with the first character being replaced with the corresponding uppercase character.

The uppercaseString getter

`uppercaseString` returns the uppercased representation of the target.

The leftSubString getter

`leftSubString` takes one int argument, `number`, and returns the sub-string from the beginning of the target and with, `number`, of characters. If the sub-string is longer than the target, the target is returned.

Example

```
let str := ["Hello World !" leftSubString : 5]
display str
```

```
str from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 85:7
    string: "Hello"
```

The rightSubString getter

`rightSubString` takes one int argument, `number`, and returns the sub-string from the end of the target and with `number` of characters. If the sub-string is longer that the target, the target is returned.

Example

```
let str := [["Hello World !" leftSubString : 11] rightSubString: 5]
display str
```

```
str from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 88:7
    string: "World"
```

The subString getter

`subString` 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 available sub-string, the sub-string is returned.

Example

```
let firstStr := ["Hello World !" subString : 6, 5]
display firstStr
let secondStr := ["Hello" subString : 10, 3]
display secondStr
let thirdStr := ["Hello" subString : 2, 10]
display thirdStr
```

```
firstStr from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 91:7
    string: "World"
secondStr from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 93:7
    string: ""
thirdStr from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 95:7
    string: "llo"
```

The `reversedString` getter

`reversedString` returns a mirrored string.

Example

```
let str := ["Hello World !" reversedString]
display str
```

```
str from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 98:7
    string: "! dlroW olleH"
```

The `componentsSeparatedByString` getter

`componentsSeparatedByString` takes one string argument: `separator`. The target is cut into string pieces according to the `separator` and a list of the pieces is returned.

Example

```
let componentList := ["Hello World !" componentsSeparatedByString : " "]
display componentList
```

```
componentList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 101:7
    list: @(
        0 :>
            string: "Hello"
        1 :>
            string: "World"
        2 :>
            string: "!"
    )
```

The `columnPrefixedBy` getter

`columnPrefixedBy` takes one string argument, `prefix`, and returns the target with each line prefixed by `prefix`.

Example

```
let formattedStr := ["Hello\nWorld" columnPrefixedBy : "# "]
println formattedStr
```

```
# Hello
# World
```

The `wrap` getter

`wrap` 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'`. It 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.

Example

```
let wrappedStr := ["Hello beautiful World.\nHow are you" wrap : 6, 2]
println wrappedStr
```

```
Hello
  beautiful
  World.
How
  are
  you
```

The `subStringExists` getter

`subStringExists` takes one argument, `subString`. It returns a bool, `true` if the sub-string `subString` is found in the target, `false` otherwise.

The `replaceString` getter

`replaceString` takes two argument, `find` and `replace`. It returns the target where each occurrence of `find` is replaced by `replace`.

The `envVar` getter

`envVar` returns a string, the value of the target environment variable. If it does not exists, `envVar` returns the empty string.

The `envVarExists` getter

`envVarExists` returns a bool, `true` if target environment variable exists, `false` otherwise.

The `string` setter

The `setCharAtIndex` setter

`setCharAtIndex` takes two arguments. The first one is the `character` to set and the second one is the `index`. It sets the character at index `index` to `character`. If `index` is greater or equal than the length of the target, a run time error occurs.

Example

```
let a := "Hello"
[!a setCharAtIndex: 'a', 1]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 111:7
    string: "Hallo"
```

The `string` functions

The following built-in functions return a string:

The `version()` function

`version()` takes no argument and returns the version string of the compiler in where GTL is embedded. For instance, in goil version 3, it returns `"3.0.0"`.

The `currentDir()` function

`currentDir()` takes no argument and returns the current directory.

The `homeDir()` function

`homeDir()` takes no argument and returns the home directory.

The `currentDateTime()` function

`currentDateTime()` takes no argument and returns the current date and time in the following format: `"<dayname> <month> <daynum> <time> <year>"`.

Example

```
println currentDateTime()
```

```
Wed Aug 17 15:16:20 2016
```

The `trueFalse()` function

`trueFalse()` is deprecated and is replaced by the `trueOrFalse` getter (see 2.10) of the `bool` data type. It takes one bool argument and returns `"true"` or `"false"` according to the argument.

The `TrueFalse()` function

`TrueFalse()` takes one bool argument and returns `"True"` or `"False"` according to the argument.

The `yesNo()` function

`yesNo()` is deprecated and is replaced by the `YESOrNO` getter (see 2.10) of the `bool` data type. This function takes one bool argument and returns `"YES"` or `"NO"` according to the argument.

The TRUEFALSE() function

`TRUEFALSE()` is deprecated and is replaced by the `TRUEOrFALSE` getter (see 2.10) of the `bool` data type. This function takes one bool argument and returns `"TRUE"` or `"FALSE"` according to the argument.

The rootTemplatesDirectory() function

This function is available in `goil`, the OIL compiler of Trampoline. `rootTemplatesDirectory()` takes no argument and return the full path of the templates directory as a string.

2.10 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`.

The bool operators

The `bool` data type supports the following operators:

Unary operator

Operator	Expression type	Meaning
~	bool ← bool	logical not
not	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

The `bool` getters

The `trueOrFalse` getter

`trueOrFalse` returns a string representation, `"true"` or `"false"`, of the bool expression.

The `string` getter

`string` returns a string representation, `"true"` or `"false"`, of the bool expression.

The `yesOrNo` getter

`yesOrNo` returns a string representation, `"yes"` or `"no"`, of the bool expression.

The `TRUEOrFALSE` getter

`TRUEOrFALSE` returns a string representation, `"TRUE"` or `"FALSE"`, of the bool expression.

The `YESOrNO` getter

`YESOrNO` returns a string representation, `"YES"` or `"NO"`, of the bool expression.

The `int` getter

`int` returns an int representation, `1` for `true` or `0` for `false`, of the bool expression.

2.11 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.

The `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

The struct getter

The map getter

`map` returns a map representation of the target.

Example

```
let a := @{ a: 1, b: 2, c: 3 }
let b := [a map]
display a
display b
```

```
a from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 120:7
    struct: @{
        a :>
            integer: 1
        b :>
            integer: 2
        c :>
            integer: 3
    }
b from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 121:7
    map: @[
        "a" :>
            integer: 1
        "b" :>
            integer: 2
        "c" :>
            integer: 3
    ]
```

2.12 The list data type

The list data type allows to store a list of data. List items are accessed by using `[index]` where *index* 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 (index 0), 2 (index 1) and 3 (index 2).

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

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

The list operators

The `list` data type supports the following operators:

Binary operators

Operator	Expression type	Meaning
+	list ← list + any	append 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 lists are equal if:

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

The `list` getters

The `length` getter

`length` returns the number of elements in the list.

The `first` getter

`first` returns the first element of the list.

The `last` getter

`last` returns the last element of the list.

The `mapBy` getter

`mapBy` takes a string argument which is the field (for an item in a list of struct) or the key (for an item in a list of a map) 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. If any of the item does not have a corresponding field/key, a run-time error occurs.

Example

```
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
```

```
myMap from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 9:7
    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"
            }
    ]
```

The set getter

`set` assumes the list is a list of items convertible to string. If any of the list item is not convertible to a string, a runtime error occurs. `set` returns a set representation of the list.

Example

```
let aList := @( 1, 2, 4, "Hello", 4, 2, 1 )
let aSet := [aList set]
display aSet
```

```
aSet from file '/Users/jlb/Develop/GTL/examples/setTest.gtl', line 30:7
    set: @!
        1, 2, 4, Hello
    !
```

The setBy getter

`setBy` takes one string argument, `fieldName`. It assumes the list is a list of struct with a field named `fieldName` convertible to a string. This field is used to build the set. If any of the list item is not a struct or does not have the field named `fieldName` or the latter is not convertible to a string, a runtime error occurs. `set` returns a set representation of the list.

Example

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

let mySet := [myList setBy: "height"]
display mySet
```

```
mySet from file '/Users/jlb/Develop/GTL/examples/listTest.gtl', line 115:7
    set: @!
        170, 175, 180
    !
```

The subListTo getter

`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 the `index` is greater than of equals the length of the target, the target is returned.

Example

```
let aList := @( 1, 2, 3, 4 )
let aList := [aList subListTo: 1]
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 125:7
    list: @(
        0 :>
            integer: 1
        1 :>
            integer: 2
    )
```

The subListFrom getter

`subListFrom` 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.

Example

```
let aList := @( 1, 2, 3, 4 )
let aList := [aList subListFrom: 2]
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 129:7
    list: @(
        0 :>
            integer: 3
        1 :>
            integer: 4
    )
```

The subList getter

`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.

Example

```
let aList := @( 1, 2, 3, 4 )
let aFirstList := [aList subList: 1, 5]
display aFirstList
let aSecondList := [aList subList: 2, 1]
display aSecondList
```

```
aFirstList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 133:7
    list: @(
        0 :>
            integer: 2
        1 :>
            integer: 3
        2 :>
            integer: 4
    )
aSecondList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 135:7
    list: @(
        0 :>
            integer: 3
    )
```

The `list` setters

The `insert` setter

`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 `index` is greater than or equals the length of the list, `data` is appended to the list.

Example

```
let aList := @( 1, 2, 3, 4 )
[!aList insert: 1, "Hello"]
[!aList insert: 10, "At the end"]
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 140:7
    list: @(
        0 :>
            integer: 1
        1 :>
            string: "Hello"
        2 :>
            integer: 2
        3 :>
            integer: 3
        4 :>
            integer: 4
        5 :>
            string: "At the end"
    )
```

2.13 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.

The **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

The **map** getters

The **length** getter

length returns an int, the number of elements in the map.

The **list** getter

list returns a list representation of the map. Elements of the list are in the alphanumerical order of the keys of the map.

Example

```
let aMap := @[ "age" : 29, "height" : 175, "name" : "John" ]
let aList := [aMap list]
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/testTypes.gtl', line 144:7
    list: @(
        0 :>
            integer: 29
        1 :>
            integer: 175
        2 :>
            string: "John"
    )
```

2.14 The **set** data type

The set data type is a set of strings. Any data that can be converted to a string may be added to a set but the actual value stored in the set is the string representation of the data. Adding a data which is already in the set has no effect.

A literal set is defined as follow.

```
@! 1, 2, 3, 4, "Hello" !
```

A literal empty set is defined as follow.

```
@! !
```

The set operators

The set data type supports the following operators:

Binary operators

Operator	Expression type	Meaning
+	set ← set + any	add the string representation of any to the set
-	set ← set_1 - set_2	remove from set_1 all the element of set_2
|	set ← set | set	does the union of sets
&	set ← set & set	does the intersection of sets

Comparison operators

Operator	Expression type	Meaning
!=	bool ← set != set	Not equal
==	bool ← set == set	Equal
>	bool ← set_1 > set_2	set_2 strictly included in set_1
<	bool ← set_1 < set_2	set_1 strictly included in set_2
>=	bool ← set_1 >= set_2	set_2 included in or equal to set_1
<=	bool ← set_1 <= set_2	set_1 included in or equal to set_2

The set getters

The length getter

length returns an int, the number of elements in the set.

Example

```
let aSet := @! 1, 2, "yes", "no" !
let len := [aSet length]
display len
```

```
len from file '/Users/jlb/Develop/GTL/examples/setTest.gtl', line 32:7
    integer: 4
```

The list getter

list getter returns a list representation of the set. Each element of the returned list is a string. Elements of the list are sorted in the alphanumerical order.

Example

```
let aSet := @! 1, "yes", 2, "no" !
let aList := [aSet list]
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/setTest.gtl', line 36:7
    list: @(
        0 :>
            string: "1"
        1 :>
            string: "2"
        2 :>
            string: "no"
        3 :>
            string: "yes"
    )
```

The `contains` getter

`contains` getter takes one argument, `item`, that must convertible to a string and returns a bool, `true` if `item` is in the set, `false` otherwise.

Example

```
let aSet := @! 1, "yes", 2, "no" !
let ok := [aSet contains: "yes"]
display ok
```

```
ok from file '/Users/jlb/Develop/GTL/examples/setTest.gtl', line 40:7
    boolean: true
```

The `elementNamed` getter

`elementNamed` getter takes one argument, `name`, that must convertible to a string and returns a string which has the same value but with the location of the element that was added to the set. If the element isnot found in the set a runtime error occurs.

The `set` setters

The `add` setter

`add` takes one argument `element` and add it to the target set. `element` must be convertible to a string. If `element` is already in the set, `add` does nothing.

The setter is redundant with the `+=` operator but more efficient because the set is not copied in the process.

Example

```
let a := @! 1, 2, "no" !
[!a add: "yes"]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/setTest.gtl', line 5:7
    set: @!
        1, 2, no, yes
    !
```

The remove setter

`remove` takes one argument, `element`, and removes it from the target set. `element` must be convertible to a string. If `element` is not in the set, `remove` does nothing.

The setter is redundant with the `+=` operator but more efficient because the set is not copied in the process.

Example

```
let a := @! 1, 2, "no" !
[!a remove: 2]
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/setTest.gtl', line 11:7
    set: @!
        1, no
    !
```

2.15 The type data type

The type data type store the type of an expression. It can be used to check dynamically the type of the arguments passed to a function, a getter or a setter, see chapter 4.

A set of constants corresponding to each type is define. These constants begin with a `'@'`: `@int`, `@char`, `@bool`, `@float`, `@string`, `@enum`, `@struct`, `@list`, `@map`, `@set` `@unconstructed` and `@type`.

To get the type of any expression, use the `type` getter. See 2.3.

Example

```
let a := 4
if [a type] == @int then
  let a += 1
end if
display a
let a := [a string]
if [a type] == @string then
  let a += "1"
end if
display a
```

```
a from file '/Users/jlb/Develop/GTL/examples/typeTest.gtl', line 14:7
    integer: 5
a from file '/Users/jlb/Develop/GTL/examples/typeTest.gtl', line 19:7
    string: "51"
```

3

GTL instructions

3.1 The `%...%` instruction

The `%...%` is the literal template string instruction. Every character appearing between the `'%'` 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.

Example

```
# Assuming we start in code mode
%This is the output of a template
%
foreach item in ["Hello World !" componentsSeparatedByString: " "]
do
  !item
between%
%
end foreach%
%
```

```
This is the output of a template
Hello
World
!
```

3.2 The `let` instruction

`let` is the variable 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	append	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	concat	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 for the first time 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 for the first time inside the `foreach` loop, its scope is set within the `foreach` loop and it does not exist after the loop anymore and `println b` will trigger and error.

3.3 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.

Example 1

```
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
```

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

Example 2

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

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

```
myStruct from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 18:7
    struct: @{
        b :>
            integer: 2
        c :>
            integer: 3
    }
```

Example 3

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

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

```
myList from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 22:7
    list: @(
        0 :>
            integer: 1
        1 :>
            integer: 2
        2 :>
            integer: 4
    )
```

Example 4

And here to remove an item from a map

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

```
myMap from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 26:7
    map: @[
        "a" :>
            list: @(
```

```
                0 :>
                    integer: 1
                1 :>
                    integer: 2
            )
    ]
```

3.4 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
```

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

3.5 The ? instruction

The `?` instruction get the current column index in the output string.

```
? var
```

Used with the **`tab`** instruction, see 3.6, `?` allows flexible formatting.

3.6 The tab instruction

The **`tab`** instruction emits spaces in the output string until the column given in argument is reached. If the current column is greater or equal than the column given in argument, no space is emitted.

```
tab expression
```

`expression` should be an `int` expression, otherwise a runtime error occurs.

3.7 The sort instruction

The **`sort`** instruction sorts a `list`. If elements of the list support operators `<` and `>` they are sorted using these two operators.

```
sort var > # descending order
sort var < # ascending order
```

Example

```
let aList := @( "wish", "you", "where", "here" )
sort aList >
display aList
sort aList <
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 30:7
    list: @(
        0 :>
            string: "you"
        1 :>
            string: "wish"
        2 :>
            string: "where"
        3 :>
            string: "here"
    )
aList from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 32:7
    list: @(
        0 :>
            string: "here"
        1 :>
            string: "where"
        2 :>
            string: "wish"
        3 :>
            string: "you"
    )
```

If the elements of the list are `struct`, a second form exists:

```
sort var by identifier > # descending order
sort var by identifier < # ascending order
```

In this case, the field *identifier* of the list item is used as sorting key.
If the field *identifier* does not exist in one of the element of the list, a runtime error occurs.

Example

```
let aList := @(
  @{ age : 18, height : 180, name : "Arnold" },
  @{ age : 22, height : 170, name : "Bob"    },
  @{ age : 29, height : 175, name : "John"   }
)
sort aList by age >
display aList
sort aList by height <
display aList
```

```
aList from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 40:7
    list: @(
        0 :>
            struct: @{
                age :>
                    integer: 29
                height :>
                    integer: 175
                name :>
                    string: "John"
            }
        1 :>
            struct: @{
                age :>
                    integer: 22
                height :>
                    integer: 170
```

```
                name :>
                    string: "Bob"
            }
        2 :>
            struct: @{
                age :>
                    integer: 18
                height :>
                    integer: 180
                name :>
                    string: "Arnold"
            }
    )
aList from file '/Users/jlb/Develop/GTL/examples/dummy.gtl', line 42:7
    list: @(
        0 :>
            struct: @{
                age :>
                    integer: 22
                height :>
                    integer: 170
                name :>
                    string: "Bob"
            }
        1 :>
            struct: @{
                age :>
                    integer: 29
                height :>
                    integer: 175
                name :>
                    string: "John"
            }
        2 :>
            struct: @{
                age :>
                    integer: 18
                height :>
                    integer: 180
                name :>
                    string: "Arnold"
            }
    )
```

3.8 The `if` instruction

`if` is the conditional execution instruction. 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 a bool. In the following example, the blue text (within the '%') is produced only if the USECOM bool variable is true:

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

3.9 The **foreach** instruction

This instruction iterates on the elements of a collection, a list, a map or a set. 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. If the collection is a set, the elements are iterated in the alphanumirical order. In all 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, map or set 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_var) 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_var*) 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: "alr2"} )

%
#define ALARM_COUNT % ![ALARMS length]

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

```
#define ALARM_COUNT 2
tpl_time_obj *tpl_alarm_table[ALARM_COUNT] = {
  &alr1_alarm_desc,
  &alr2_alarm_desc
};
```

3.10 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.

3.11 The loop instruction

The `loop` instruction iterates 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
  println a
end loop
```

```
0
2
4
6
8
10
```

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

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

```
25
24
23
22
21
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 $2^{32} - 1$

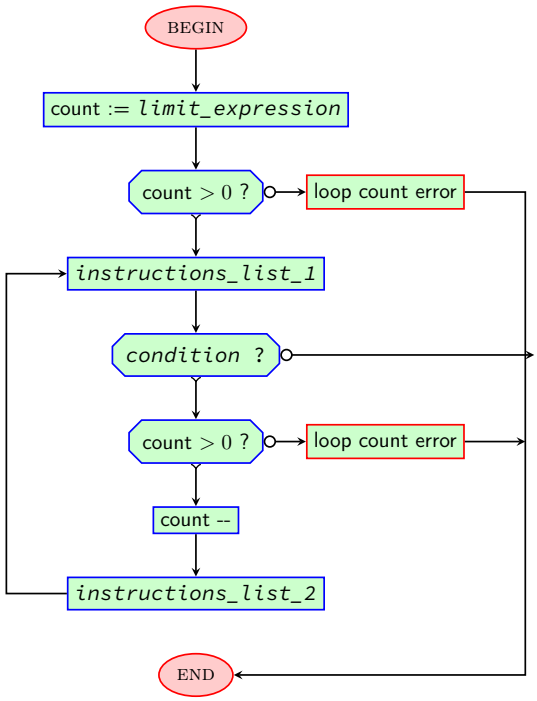

Figure 3.1: semantics of the **repeat** instruction

3.12 The **repeat** instruction

The **repeat** instruction combines 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_expression*, 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 3.1

Example

```
let aList := @( 1, 2, 3, 4 )
repeat ( [aList length] )
while [aList length] > 0 do
  println [aList first]
  unlet aList[0]
end repeat
```

```
1
2
```

```
3
4
```

3.13 The write instruction

The **write** instruction writes the template output string to a file. The general form is:

```
write to <executable> expression :
  instruction_list
end write
```

Where *expression* is a string expression. Here the template output string built by instructions in the *instruction_list* are written to the file named after the evaluation of *expression*.

If the optional keyword **executable** is present, the file has its executable bit set.

Example 1

```
write to "/tmp/outputOfTemplate" :
%Hello
%
end write
```

```
Created '/tmp/outputOfTemplate'.
```

The content of file at path '/tmp/outputOfTemplate' is:

```
Hello
```

Example 2

```
write to executable "/tmp/listfiles.sh" :
%# /bin/sh
ls -al
%
end write
```

```
Created '/tmp/listfiles.sh'.
```

The content of file at path '/tmp/outputOfTemplate' is:

```
# /bin/sh
ls -al
```

And since it is an executable file, one can type '/tmp/listfiles.sh' to execute the shell script.

3.14 The template instruction

The **template** instruction invokes another template. The output string built by the invoked template is included in the output string of the caller at the location occupied by the **template** instruction. The general form is:

```
template <(argument_list)> <if exists> file_reference <in hierarchy>
<or>
  <instruction_list>
<end template>
```

Template invocation with a copy of variables

The simplest form is:

```
template file_reference
```

The `file_reference` may be an identifier or **`from`** followed by an expression that evaluate to a `string`.

```
template aTemplate # identifier file reference
```

or

```
template from "aTemplate" # string file reference
```

The second one allow to use any file name. In both cases, the **`template`** instruction looks for a file named `aTemplate.extension`. `extension` is customizable. For *gtl* it is set to `"gtl"`. For *goil* it is set to `"goilTemplate"`.

If the template file is found it get a copy of the variables of the caller and is executed. If it is not found a runtime error occurs.

Example
Contents of `callerTemplate.gtl`:

```
loop a from 1 to 4 do
  template helloTemplate
  % % !a % %
end loop
%
%
```

Contents of `helloTemplate.gtl`:

```
Hello
```

```
Hello 1 Hello 2 Hello 3 Hello 4
```

Template invocation with arguments

Instead of passing a copy of all the variables to the invoked template, it is possible to pass arguments. In this case the invoked template works as a procedure and has no access to the variables of the caller. See 3.15 too.

Example
Contents of `caller2Template.gtl`:

```
loop a from 1 to 4 do
  template (a) hello2Template
end loop
%
%
```

Contents of `hello2Template.gtl`:

```
input(number)
%Hello % !number % %
```

```
Hello 1 Hello 2 Hello 3 Hello 4
```

Conditional invocation of templates

In some cases, the invocation of a template may be optional and if the template is not found, no error should occurs. The following form goes this:

```
template if exists file_reference
```

If the template file is found it get a copy of the variables of the caller and is executed. If it is not found the template invocation fails silently.

When the template file is not found it is possible to execute a list of instructions. This is done by adding a `or`, the instruction list and `end template`.

```
template if exists file_reference or
  instruction_list
end template
```

3.15 The input instruction

The `input` instruction is used in a template to retrieve the arguments passed to the template by the caller. See section 3.14

```
input(formal_argument_list)
```

`input` may be used at any time in the template. For each variable appearing in the *formal_argument_list* input pops the first value from the argument list passed by the caller. This can be done by one or more `input` instructions.

Example

```
# The caller invoke a template with argument 1, 2, 3 and 4
template (1, 2, 3, 4) aTemplate
```

Contents of file `aTemplate.gtl`:

```
input(a)    # retrieve 1
input(b, c) # retrieve 2 in b and 3 in c
input(d)    # retrieve 4
input(e)    # trigger a runtime error, the argument list is empty
```

Arguments in the *formal_argument_list* may be typed. In the following example any data type may be passed for `d` but `c` requires an `int` and `a` requires a `string`.

Example

```
template (3, 2, 1) aTemplate
```

Contents of file `aTemplate.gtl`:

```
input(c : @int, d) # retrieve 3 in c and 2 in d
input(a : @string) # trigger an error int 1 is not a string
```

3.16 The `error` and `warning` instructions

It can be useful to generate an error or a warning if a data is not defined or if its value is unappropriate. `error` and `warning` have 2 forms:

```
error var : expression
warning var : expression
```

or

```
error here : expression
warning here : expression
```

expression must be of type `string`. In the first form, *var* is a variable. The file location of this variable may be a location in any input file of the compiler which embeds the GTL interpreter or in the template file if the variable was assigned in the template. This location is used to signal the location of the error or warning. In the second form, `here` means the current location in the template file.

In the following example taken from the Goil templates, an error is generated if the `ACTIVATION` attribute of an extended task is greater than `1`:

Example 1

```
# Check no extended task as an ACTIVATION attribute greater than 1
foreach task in EXTENDEDTASKS do
  if task::ACTIVATION > 1 then
    error task::ACTIVATION : "An extended task cannot have ACTIVATION greater than 1"
  end if
end foreach
```

In this second example, a warning is generated if a template is not found:

Example 2

```
template if exists interrupt_wrapping or
  warning here : "interrupt_wrapping.goilTemplate not found"
end template
```

3.17 The `print` and `println` instructions

`print` an `println` print an *expression* to the standard output. `println` prints a `'\n'` after the expression. `println` may be used alone to print a `'\n'`.

```
print expression
println <expression>
```

Any `string`, `int`, `bool`, `float`, `enum`, `type` and `char` may be printed. `struct`, `list`, `map` and `unconstructed` may not.

Example

```
foreach a in @( "Does", "anybody", "remember", "Vera", "Lynn", '?' ) do
  print a
  print " "
end foreach
println
```

```
Does anybody remember Vera Lynn ?
```

3.18 The **display** instruction

display prints the name of any variable, the location of the **display** instruction and the content of any variable. It is designed to be use for debug purpose. To get the following output, a **display** TASKS has been added to `root.goilTemplate` and the example in `examples/cortex/-armv7em/stm32f407/stm32f4discovery/alarms` has been compiled by Goil.

Example

```
TASKS from file '/Users/jlb/Develop/trampoline-git-maintain/goil/templates/root.
    goilTemplate', line 1467:7
    list: @(
        0 :>
            struct: @{
                ACTIVATION :>
                    integer: 1
                AUTOSTART :>
                    boolean: false
                KIND :>
                    string: "Task"
                NAME :>
                    string: "blink"
                NONPREEMPTABLE :>
                    boolean: false
                PRIORITY :>
                    integer: 1
                SCHEDULE :>
                    string: "FULL"
                STACKSIZE :>
                    integer: 300
                USEFLOAT :>
                    boolean: false
                USEINTERNALRESOURCE :>
                    boolean: false
            }
        1 :>
            struct: @{
                ACTIVATION :>
                    integer: 1
                AUTOSTART :>
                    boolean: false
                KIND :>
                    string: "Task"
                NAME :>
                    string: "read_button"
                NONPREEMPTABLE :>
                    boolean: false
                PRIORITY :>
                    integer: 2
                SCHEDULE :>
                    string: "FULL"
                STACKSIZE :>
                    integer: 300
                USEFLOAT :>
                    boolean: false
                USEINTERNALRESOURCE :>
```

```
                    boolean: false
            }
    )
```

3.19 The **variables** instruction

variables displays all the variables. It is designed to be use for debug purpose.

```
variables
```

Example

```
let a := @( 1, 2, 3 )
let b := @{ x:1, y:2, z:3 }
let c := @[ "age": 10, "name": "Vera" ]
let d := "Hello"
let e := 3
variables
```

```
================= Variables ================== Displayed from =================
file '/Users/jlb/Develop/GTL/examples/variables.gtl', line 7:9
================================================================================
--------------------------------------------------------------------------------
a
--------------------------------------------------------------------------------
list: @(
    0 :>
        integer: 1
    1 :>
        integer: 2
    2 :>
        integer: 3
)
--------------------------------------------------------------------------------
b
--------------------------------------------------------------------------------
struct: @{
    x :>
        integer: 1
    y :>
        integer: 2
    z :>
        integer: 3
}
--------------------------------------------------------------------------------
c
--------------------------------------------------------------------------------
map: @[
    "age" :>
        integer: 10
    "name" :>
        string: "Vera"
]
--------------------------------------------------------------------------------
d
--------------------------------------------------------------------------------
string: "Hello"
--------------------------------------------------------------------------------
```

```
e
------------------------------------------------------------------------------
integer: 3
==============================================================================
```

3.20 The **seed** instruction

seed sets the random seed of the random number generator. If given, its argument should evaluate to an integer and should be within 0 and $2^{32} - 1$. If not given the seed is set by summing the hours, minutes, seconds of the current date.

4

GTL modules

GTL may be extended with functions, setters and getters. Definitions of these objects are done in separates `.gtm` files called modules.

4.1 Importing a module

Modules are imported by using the **`import`** statement. GTL prevents multiple import of the same module.

```
import expression
```

`expression` must evaluate to a `string`. **`import`** is not an instruction and must appear at the beginning of the template file before any instruction except the `%...%` instruction.

4.2 Writing a module

A module includes zero or more functions definitions, zero or more getter definitions and zero or more setter definitions. It may include **`import`** statements too but they must appear before any function / getter / setter definition. If `%...%` instructions are used they are ignored. Other instructions which output data in the output template string are forbidden.

The arguments

Arguments are passed by copy. Each formal argument of the list may be typed or not. If a formal argument is typed, GTL emits a runtime error if the passed argument does not have the same type. Here is a formal argument list with the second argument typed.

```
a, b : @int, c
```

Any data type may be passed for arguments `a` and `c` but an `int` is required for argument `c`. The type of an argument may be tested, check 2.15, 2.3 and 4.2.

Function definition

A function definition has the following form:

```
func func_name(formal_argument_list) result_var
  instruction_list
end func
```

The `formal_argument_list` may be empty. `result_var` is a local variable of the function which is used to return the result.

Example 1

Content of `function.gtm`:

```
func square(x) result
  let result := x * x
end func
```

Content of `template.gtl`:

```
import "function"

let b := square(6)
display b
let b := square(2.6)
display b
```

```
b from file '/Users/jlb/Develop/GTL/examples/template.gtl', line 5:7
    integer: 36
b from file '/Users/jlb/Develop/GTL/examples/template.gtl', line 7:7
    float: 6.76
```

Example 2

Content of `function.gtm`:

```
func square(x : @int) result
  let result := x * x
end func
```

Content of `template.gtl`:

```
import "function"

let b := square(6)
display b
let b := square(2.6)
display b
```

```
b from file '/Users/jlb/Develop/GTL/examples/template.gtl', line 5:7
    integer: 36
/Users/jlb/Develop/GTL/examples/template.gtl:6:17:19:
semantic error #1: int expected for x
let b := square(2.6)
----------------^^^
```

Example 3

Content of `function.gtm`:

```
func square(x) result
  if [x isANumber] then
    let result := x * x
  else
    error here : "int or float expected"
  end if
end func
```

Content of `template.gtl`:

```
import "function"

let b := square(6)
display b
let b := square(2.6)
display b
let b := square("Hello")
display b
```

```
b from file '/Users/jlb/Develop/GTL/examples/template.gtl', line 5:7
    integer: 36
b from file '/Users/jlb/Develop/GTL/examples/template.gtl', line 7:7
    float: 6.76
/Users/jlb/Develop/GTL/examples/template.gtl:6:18:24:
semantic error #1: int or float expected
let b := psquare("Hello")
-----------------^^^^^^^
```

4.3 Getter definition

A getter is defined as follow:

```
getter a_type getter_name(formal_argument_list) result_var
  instruction_list
end getter
```

a_type designates the data type on which the getter apply. It can be `@int`, `@char`, `@float`, `@bool`, `@enum`, `@string`, `@struct`, `@list`, `@map`, `@type` or `@unconstructed`. *getter_name* is the name of the getter. The same name may be used for several types. The *formal_argument_list* follows the same rules as those exposed in 4.2. *result_var* is the name of the variable used to store the result of the getter.

Within the getter, variable **self** is a copy of the variable targeted by the getter. It can be modified but the target will not be touched.

Example 1

In this example a getter to get the square of an `int` is defined

Content of `getter.gtm`:

```
#-----------------------------------------------------------------
# getter : square of a int
#-----------------------------------------------------------------
getter @int square() result
  let result := self * self
end getter
```

Content of `template.gtl`:

```
import "getter"

let b := [6 square]
display b
let b := [b square]
display b
```

```
b from file '/Users/jlb/Develop/GTL/examples/testGetter.gtl', line 5:7
    integer: 36
b from file '/Users/jlb/Develop/GTL/examples/testGetter.gtl', line 7:7
    integer: 1296
```

Example 2

In this example two getters, one to check a `struct` and the other to check a `list` are defined
Content of `getter.gtm`:

```
#-----------------------------------------------------------------
# getter : returns true if the target has a field
#-----------------------------------------------------------------
getter @struct hasField(field : @string) result
  let selfAsMap := [self map]
  let result := exists selfAsMap[field]
end getter


#-----------------------------------------------------------------
# getter : returns true if all the elements of the list are
# structs and if all the elements of the list have a field
#-----------------------------------------------------------------
getter @list alwaysHasField(field : @string) result
  let result := true
  foreach element in self do
    let result &= [element hasField : field]
  end foreach
end getter
```

Content of `template.gtl`:

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

if [c alwaysHasField : "age"] then
  println "List 1 ok"
else
  println "List 1 ko"
end if

let c += @{ height : 150, name : "Sally" }

if [c alwaysHasField : "age"] then
  println "List 2 ok"
else
  println "List 2 ko"
end if
```

```
List 1 ok
List 2 ko
```

4.4 Setter definition

A setter is defined as follow:

```
setter a_type setter_name(formal_argument_list)
  instruction_list
end getter
```

`a_type` designates the data type on which the setter apply. It can be `@int`, `@char`, `@float`, `@bool`, `@enum`, `@string`, `@struct`, `@list`, `@map`, `@type` or `@unconstructed`. `setter_name` is the name of the setter. The same name may be used for several types. The `formal_argument_list` follows the same rules as those exposed in 4.2.

Within the setter, variable **`self`** is the variable targeted by the setter.

Example

Content of `setter.gtm`:

```
#-------------------------------------------------------------------
# setter : delete all list items which are struct and with a
# field named age
#-------------------------------------------------------------------
setter @list deleteAge()
  let theList := self
  let offset := 0
  foreach element (index) in theList do
    if exists element::age then
      unlet self[index + offset]
      let offset -= 1
    end if
  end foreach
end setter
```

Content of `template.gtl`:

```
let c := @(
  @{ age : 18, height : 180, name : "Arnold" },
  @{ age : 22, height : 170, name : "Bob"    },
  @{ age : 29, height : 175, name : "John"   },
  @{           height : 150, name : "Sally"  }
)

[!c deleteAge]
display c
```

```
c from file '/Users/jlb/Develop/GTL/examples/template.gtl', line 36:7
    list: @(
        0 :>
            struct: @{
                height :>
                    integer: 150
                name :>
                    string: "Sally"
            }
    )
```

5

The GTL debugger

GTL includes a debugger. To launch the debugger, invoke GTL with the `--debug` command line option. The GTL program stops before executing the first instruction and the debugger displays its prompt:

```
Starting debugger, type help for command list
gtl>
```

The debugger includes commands to display variables and expression, to set breakpoints and watchpoints, to execute the GTL program step by step and to execute a set of commands each time an instruction is executed in step by step mode.

5.1 The help command

The help command displays a help text:

```
Starting debugger, type help for command list
gtl> help
Available commands:
  break <filename>:<line>        : set a breakpoint at <line> in file <filename>
  break <line>                   : set a breakpoint at <line> in the current file
  break                          : lists the breakpoints
  break not <num>                : delete breakpoint at index <num>
  cont                           : continue execution until the next breakpoint or the end
  display <variable>             : display variable <variable>
  do <command>                   : do a command each time a step is done
  do                             : list the do commands
  do not <num>                   : delete the do command at index <num>
  list                           : lists instructions +/- 5 around current one
  list <num>                     : lists instructions +/- <num> around current one
  let <variable> := <expression> : compute <expression> and set <variable> to the result
  print <expression>             : prints the <expression>
  step                           : step one instruction
  unlet <variable>               : delete <variable>
  variables                      : display all variables in scope
  watch (<expression>)           : set a watchpoint matching the boolean <expression>
  watch                          : lists the watchpoints
  watch not <num>                : delete watchpoint at index <num>
  <return>                       : step one instruction
```

6

Using GTL in a GALGAS project

It is quite simple to include GTL in a GALGAS project. In this chapter we will list the files you will use, how to invoke a template and how to populate the variables with data.

6.1 Needed files

The following files from the standalone GTL project are needed:

File	Content
gtl_types.galgas	Internal types of GTL
gtl_expressions.galgas	Expression classes
gtl_data_types.galgas	Types of GTL
gtl_scanner.galgas	Lexical analyzer
gtl_parser.galgas	Instructions parser for the template files
gtl_instruction_parser.galgas	Instruction parser common to template and module files
gtl_module.galgas	Module classes
gtl_module_parser.galgas	Instruction parser for the module files
gtl_module_grammar.galgas	Grammar of a module file
gtl_expression_parser.galgas	Expression parser for template and module files
gtl_instructions.galgas	Instruction classes
gtl_grammar.galgas	Grammar of a template file
gtl_functions.galgas	Built-in functions
gtl_options.galgas	Options of GTL

Add them in your `galgas-sources` directory and reference them in your project file.

6.2 Invoking a template

The best way to invoke a template is to use the `invokeGTL` function. `invokeGTL` takes 3 arguments and returns the output string of the template. Arguments are:

The `rootTemplateFileName` argument

This argument is the file name of the GTL template to invoke. Its type is `@gtlString`. If your file name is in a `@lstring` use the `lstringToGtlString` function to build the `@gtlString`. If your file name is in a `@lstring` use the `stringToGtlString` function.

The `context` argument

This argument defines the execution context of GTL. It is an instance of the class `@gtlContext`, or a derived class if you want to customize it, that gathers the data of the context. Function `emptyContext` may be used to get a default context that may be completed after. The `@gtlContext` attributes may be changed by using the following setters