Introduction to Remopla

Contents

1 Foreword

The Remopla language, an input language of the Moped model checker, is aimed to modelling behaviours of finite-domain programs with (possibly) unbounded recursive procedure calls. The model checker can perform reachability analysis of the models in Remopla. To be more precise, the Moped model checker implements reachability analysis of symbolic push down systems and the Remopla language provides a way to describe symbolic push down systems in terms of programs with recursive procedure calls.

This document gives a terse description of the syntax and semantics of the language.

2 Terminology of Language Elements

A Remopla code implementing behaviour of a program will be called a *model* throughout this document because there are some substantial differences between a real program and its model in Remopla. For instance, there is basically no way to execute a model and observe a specific run of it. The primary purpose of a model in Remopla is to allow for queries on reachability of check points of the model, so it rather describes possible directions of execution than implements a deterministic algorithm.

There are three distinct syntax elements a model in Remopla consists of at the top level: definitions, declarations and statements.

A *definition*, being it a constant definition or a module definition, is not terminated by a semicolon. The semantics and exact syntactic rules determining the extent of a constant definition and a module definition are discussed in sections 4 and 9, respectively.

Each *declaration* is terminated by a semicolon. The Remopla language supports data type and variable declarations (see section 5), module declarations (see section 9) and a declaration of initial configuration. The declaration of initial configuration is somewhat special to the Remopla language (see section 6 for details).

Each *statement* is terminated by a semicolon, too. A statement can be a skip statement (see section 8.1), a go-to statement (see section 8.2), an assignment statement (see section 8.3), a break statement (see section 8.4), a conditional statement, a loop statement (see section 8.5 and a module call and a return statement (see section 9).

Any statement can be given a *label*. Labelling a statement makes it possible to bring control flow to that statement by a go-to statement and to query reachability of the label.

Types, variables, modules, labels and elements of enumerated data types are named by *identifiers*. An identifier is any non-empty string of alphanumeric characters starting with a letter which is not a keyword of Remopla.

The keywords will be introduced during the next sections. See appendix 10.1 for the complete list of keywords of the Remopla language.

Remopla is a case sensitive language. The case matters for both keywords and identifiers.

The rules governing name spaces of identifiers are rather complex. To stay on the safe side, one should not use a single identifier to denote two different entities. For instance, using an identifier for a constant and anything else would produce a syntax error. Using an identifier as a module name and as a label would work just fine but the semantics would differ from what one might have expected.

3 Model Structure

Any model written in Remopla has to follow the structure as follows. See example 1 for a very simple example demonstrating the top-level structure of a model.

⟨constant definitions⟩
⟨global declarations⟩
⟨initial configuration⟩
⟨model body⟩

Constant definitions must come at the very beginning of a model. They are not allowed anywhere else. See section 4 for details regarding constant definitions.

Global declarations stand for type declarations, module declarations and declarations of global variables. Again, these declarations cannot be placed anywhere else. Type declarations and declarations of variables are covered in section 5. See section 9 for more information on module declarations.

Initial configuration can be thought of as the last global declaration—it is mandatory for any model and it separates the section of global declarations from the model body. The declaration specifies the configuration from which the forward reachability analysis of the model starts or by which the backwards reachability analysis is terminated. The details are given in section 6.

Model body consists of statements and module definitions implementing the behaviour of the model. The statements are usually organized in modules but it is not necessary. Stand-alone statements are commonly used for infinite labelled loops defining the check points the reachability of which is to be queried. Line 12 of example 1 demonstrates such a loop.

```
1   # constant definitions
2   define DEFAULT_INT_BITS 4
3
4   # global declarations
5   module void main();
6
7   # initial configuration
8   init main;
9
10  # model body
11
12  error: goto error;
13
14  module void main () {
15      goto error;
16  }
```

Remopla Example 1: The Simplest Model (`simplest.rem`)

4 Constant Definitions

A constant in Remopla is a symbolic name for a non-negative integer. The primary use of constants is to parametrize models in the sense that a Remopla code can implement many instances of a problem depending on the values of constants. Every constant a model uses must be defined at the very beginning of the model. The syntax of a definition is as follows.

`define` ⟨identifier⟩ ⟨constant integer expression⟩

Every constant definition starts with a keyword `define` and is terminated by the next occurrence of the keyword or by any global declaration including the declaration of initial configuration. The second argument of the definition must be a constant integer expression (see section 7.1), i.e. an integer expression which combines numbers and previously defined constants only.

4.1 Internal Constants

Internal constants parametrize the process of reachability analysis. Regarding syntax, internal constant is a keyword which can be used instead of identifier in a constant definition and nowhere else.

There is only one internal constant at the moment but there are more to come. The one is denoted by a keyword `DEFAULT_INT_BITS` and does exactly what the name suggests: it sets the default bit size of integers. For instance, the default bit size of integers in example 1 would be four. See section 5.2 for more information on integer bit sizes.

5 Data Types and Variable Declarations

The Remopla language supports boolean, unsigned integer and enumerated data types. Moreover, structured data types and one- or two-dimensional arrays can be constructed from the basic types. This section concerns declarations of data types and variables. Formal parameters of modules and return values of modules are treated in section 9.

5.1 Booleans

Declarations of boolean variables start with a keyword `bool` which is followed with a comma-separated list of identifiers denoting the names of variables being declared.

`bool` ⟨variable⟩`, ...,` ⟨variable⟩`;`

The value of a boolean variable can be either `true` or `false` where the two keywords have the obvious meaning. Boolean variables and the two keywords form the atomic boolean expressions from which compound ones can be constructed (see section 7.2).

5.2 Integers

The Remopla language supports unsigned integer variables of limited range. Integers are implemented as bit arrays where the least significant bit is the rightmost one. Every integer variable must be given its bit size, i.e. the length of the bit array storing its value. Assuming that the length of the bit array storing a variable is n, the value of the variable can range from 0 to $2^n - 1$.

A default bit size can be given by setting the `DEFAULT_INT_BITS` internal constant (see section 4.1). The bit size of each variable must be specified in the declaration of the variable if the default is not given.

The syntax of integer variable declarations is as follows. The declaration starts with a keyword `int` which is followed with a comma-separated list of variable specifications. A variable specification consists of an identifier followed optionally with a parenthesized bit size. The bit size is given as a constant integer expression. It is not optional if the default bit size is not given.

`int` ⟨variable⟩`,` ⟨variable⟩`(`⟨bits⟩`), ...,` ⟨variable⟩`;`

For instance, the next snippet of Remopla code declares a variable `area` of bit size 8 and variables `width` and `height`, both of bit size 4.

```
define DEFAULT_INT_BITS 4
int area(8), width, height;
```

The rules governing evaluation of integer expressions involving variables of different bit sizes are covered in section 7.1.

5.3 Enumerations

In general, an enumeration is a finite set of user-defined elements. The elements are named with identifiers. There are two forms of declarations concerning enumerated data types. The first form, which involves definition of the new data type, is as follows.

```
enum ⟨type name⟩ {
    ⟨element⟩, ..., ⟨element⟩
} ⟨variable⟩, ..., ⟨variable⟩;
```

The declaration starts with a keyword `enum`. The keyword is followed with an optional identifier denoting the name of the new data type. Then comes a non-empty list of elements of the data type in braces. The declaration ends with a comma-separated list of identifiers of declared variables. The list may be empty.

It is impossible to declare a variable of the same enumerated data type later on if the type name is omitted. If the type name is present, one can use the second form of declaration to declare more variables of the enumerated data type later on.

```
enum ⟨type name⟩ ⟨variable⟩, ..., ⟨variable⟩;
```

The type name cannot be omitted and the list of variables cannot be empty in this form of declaration.

Strictly spoken, the use of variables of enumerated data types should restrict to assignments and tests on equality within the same enumerated type. However, the implementation allows to misuse them in many ways so do not be surprised by the lack of "syntax error" reports.

5.4 Structures

Structured variables are declared in a similar fashion as variables of enumerated data types. There are two forms of declaration again. The first one, which involves definition of a new data type, is as follows.

```
struct ⟨type name⟩ {
    ⟨item⟩;
    ...
    ⟨item⟩;
} ⟨variable⟩, ..., ⟨variable⟩;
```

The declaration starts with a keyword `struct`. The keyword is followed with an optional identifier denoting the name of the new data type. Then comes the definition of the structure in braces. The declaration ends with a comma-separated list of declared variables. The list may be empty.

The body of the structure is a semicolon-separated list of declarations of inner variables. The declarations do not differ from the ones at the top level. They may even involve formerly declared enumerated and structured data types as well as declarations of new ones to arbitrary depth. However, recursive declarations, i.e. declarations of variables of the structured data type currently being defined, are not allowed.

If a declaration contains the type name, the name can be used to declare variables of the same structured type later on. The second form of declarations takes place in that case.

`struct` ⟨type name⟩ ⟨variable⟩`, ...,` ⟨variable⟩`;`

The type name cannot be omitted and the list of variables cannot be empty in this form of declaration.

The only actual use of variables of structured data types is to make passing local data to and from modules easier (see section 9). Variables of a structured data type cannot be used anywhere else. Of course, one can use the inner variables of a structured variable provided they are of simpler data types (and there eventually are some). They are accessed via the "dot" notation as follows.

⟨variable⟩`.`⟨inner var⟩`. ... .`⟨inner var⟩

For instance, consider the declaration of a structured variable `rct` below. One can write `rct.dims.width` to access the width of the rectangle `rct`.

```
struct rectangle {
    struct {
        int width, height;
    } dims;
    int area(8);
};
struct rectangle rct;
```

5.5 Arrays

A variable of any type can also be declared as a one-dimensional or two-dimensional array the elements of which are of that type. This is done by attaching an array specification to the identifier of the variable being declared. The array specification can be of any of the following forms.

⟨identifier⟩ `[`⟨n⟩`]`
⟨identifier⟩ `[`⟨m⟩`,`⟨n⟩`]`
⟨identifier⟩ `[`⟨n⟩`][`⟨q⟩`]`
⟨identifier⟩ `[`⟨m⟩`,`⟨n⟩`][`⟨p⟩`,`⟨q⟩`]`

The first two forms declare a one-dimensional array. The last two forms declare a two-dimensional array. If a dimension specification consists of a single positive integer, for instance n, that dimension of the array has n elements indexed from 0 to $n-1$. If it consists of two nonnegative integers where the first one is lower than or equals the second one, for instance m and n where $m \leq n$, then that dimension of the array has $n-m+1$ elements indexed from m to n.

Constant integer expressions (see section 7.1) can be supplied instead of plain numbers in dimension specifications.

For example, consider the following declarations.

```
int a1[3], a2[2,4][1,4](3);
bool b1[0,4], b2[5];
```

Arrays `b1` and `b2` are essentially the same. They both have five boolean elements indexed from 0 to 4. The array `a1` has three integer elements indexed from 0 to 2, each one being of the default bit size. The array `a2` is a two-dimensional array. It consists of 12 integer elements organized in a 3×4 matrix, each element being of bit size 3.

6 Initial configuration

Global declarations in a model are always terminated by a mandatory declaration of initial configuration. The basic form of that declaration has syntax

`init` ⟨name⟩`;`

where name can be a module name or a label. The meaning of such a statement is that forward reachability analysis starts at the statement with the given label or with execution of the given module. If the module takes parameters, all possible values are taken into consideration for the reachability analysis.

There is also a more complex form of this declaration which is beyond the scope of this document.

7 Expressions

The Remopla language distinguishes three types of expressions: constant integer expressions, general integer expressions and boolean expressions. Regarding syntax, constant integer expressions comprise a strict subset of

general ones which is why we are going to explain them simultaneously in a single subsection. Keep in mind, however, that there are substantial differences between the two when it comes to evaluation.

7.1 Integer Expressions

Integer expressions of Remopla are built up from integers in decimal representation, constants and integer variables using operators and parentheses. The operators are listed below with decreasing precedence. All operators are binary. Precedence can be forced by parentheses.

`* /`	integer multiplication and division
`+ -`	integer addition and subtraction
`<< >>`	left and right bit shift
`&`	bitwise conjunction
`^`	bitwise exclusive disjunction
`|`	bitwise disjunction

The three bitwise operators are not implemented for general integer expressions yet.

A *constant integer expressions* is an expression which contains no integer variable as its subexpression. Bitwise operators are implemented for constant integer expressions. Of course, constant integer expressions defining the value of a constant may involve previously defined constants only.

The type of an integer expression is determined by its usage in Remopla code. If the syntax of Remopla allows for constant expressions only, it is an error to use a general expression. On the other hand, if the syntax allows for general expressions, all expressions are treated as general ones even if they are constant by syntax.

7.1.1 Evaluation of Constant Integer Expressions

Constant integer expressions are evaluated in a special manner compared to the evaluation of general ones. In fact, they are evaluated during parsing-time. The limits on values of integer constants and the behaviours of models are architecture dependent as a result. Most architectures use 32-bit implementation of integers which should be pretty enough for any model in Remopla because reachability analysis of models with large data is intractable.

7.1.2 Evaluation of General Integer Expressions

A general integer expression can be used either in assignment to an integer variable or in comparison as a part of a boolean expression. It is evaluated per se in neither case. Both cases should be better viewed as relations.

Let us consider assignments first (see also section 8.3). In general, an assignment can be made for any input values of variables on the right hand side for which the expression yields a value that fits into the range of the variable at the left hand side of the assignment. And this is exactly what happens. The size of no subexpression matters. If the expression value exceeds the bounds of the left hand side variable, the assignment simply does not take place and, in terms of execution of a program, the execution would be aborted. In terms of reachability, there is no reachable configuration beyond a configuration for which an assignment is not possible.

The relational view is even more straightforward for comparisons of two integer expressions because the comparison itself puts variables of the two expressions into a relation. The comparison is then satisfied exactly for those values of variables which belong to the relation. Again, potential size of no subexpression matters.

7.2 Boolean Expressions

The primary use of boolean expressions in Remopla is to guard clauses of conditionals and loops (see section 8.5). Of course, the expressions are also used in assignments to boolean variables (see sections 5.1 and 8.3).

Boolean expressions of Remopla are built up from keywords `true` and `false`, boolean variables and comparisons of integer expressions which are combined using parentheses and boolean operators. Comparison operators `<`, `<=`, `==`, `!=`, `>=` and `>` with the common meaning are available for integer expressions.

Two boolean expressions can be combined using binary operators of logical disjunction (`||`), logical conjunction (`&&`) and logical equivalence (`<=>`). The operators are listed with decreasing precedence.

The unary logical operators have the highest priority of all operators in Remopla. There is a logical negation operator (`!`) and universal and existential quantifiers. The quantifiers are denoted by keywords `A` and `E`, respectively, and their usage must follow the syntax as follows.

`A` ⟨identifier⟩ `(`⟨m⟩`,`⟨n⟩`)` ⟨boolean expression⟩
`E` ⟨identifier⟩ `(`⟨m⟩`,`⟨n⟩`)` ⟨boolean expression⟩

A quantified expression starts with a keyword denoting the quantifier. Then comes an identifier and the range of its values given as two constant integer expressions m and n where $m \leq n$. The identifier can be used as a read-only integer variable of the given range in the boolean expression that follows.

A universally quantified expression holds if and only if the inner expression holds for every value of the quantified variable. An existentially quantified expression holds if and only if the inner expression holds for at least one value of the quantified variable.

The quantified expressions can be very handy when it comes to tests on arrays.

8 Statements

We are not going to give formal semantics of statements in this section. We will proceed with precise enough informal description. However, some formal basic can be useful for better understanding.

The Remopla language is aimed to implementing models of programs the behaviours of which are then subject of reachability analysis. The analysis is performed on configurations of the model in question and transitions between them. A configuration of a model consists of instruction counter (determining the next statement to be executed), current values of global variables and a call stack. Transitions between configurations are defined by the statements determined by the instruction counter of the configurations. For each configuration its "next" statement determines to which configurations the model can move.

It is perfectly possible that, for a given configuration and a statement, the model can move nowhere or, in other words, the statement cannot be executed. It just means that there is no reachable configuration from the one in question.

8.1 Skip Statement

The skip statement has syntax as follows

`skip` ⟨boolean expression⟩`;`

where the boolean expression is optional. The form of the statement without the expression is usually used to fulfil syntax requirements of clauses of conditionals and loops. It simply does nothing and passes the control to the next statement.

The form of the statement with the expression is more tricky. The statement "does nothing" and passes the control to the next statement if and only if the expression holds. The execution aborts otherwise, i.e. there are no transitions from configurations for the data of which the expression does not hold. Consider the following snippet of Remopla code.

```
skip false; lbl: skip;
```

The label *lbl* is not reachable in this case.

8.2 Go-To Statement

We have already mentioned in section 2 that any statement can be given a label. The go-to statement can be used to move the control to any labelled

statement without affecting data or call stack. The syntax of the statement is as follows.

```
goto ⟨label⟩;
```

8.3 Assignments

Assignments are at the core of any model. Let us start with the simplest form of assignments which assigns a value of an expression to an integer variable or a boolean variable.

```
⟨variable⟩ = ⟨expression⟩;
```

Of course, the expression has to be of the same type as the variable. An assignment of a boolean expression to a boolean variable is always executed and the control passes to the next statement. It is not the case for assignments to integer variables. Integer variables are bounded and the value of an expression can exceed the bounds of the variable to which it should be assigned. The execution aborts in that case. In the terms of configurations, there are no next configurations for the ones in which the value of the expression exceeds the bounds of the variable. For instance, the label *lbl* is not reachable in the following snippet of Remopla code.

```
int n(3);
n = 4;
n = n*n;
lbl: skip;
```

Several assignments can be put in parallel in the Remopla language.

```
⟨variable⟩ = ⟨expression⟩, ..., ⟨variable⟩ = ⟨expression⟩;
```

All the expressions are "evaluated" (see section 7.1 for particularities of integer expression evaluation) according to current values of variables first and then the assignment takes place. The statement is not executed if any of the assignments cannot be done because of exceeded bounds.

There is yet more to assignments. An assignment can be quantified in a similar way as a boolean expression. The syntax is as follows

```
A ⟨identifier⟩ (⟨m⟩,⟨n⟩) ⟨variable⟩ = ⟨expression⟩
E ⟨identifier⟩ (⟨m⟩,⟨n⟩) ⟨variable⟩ = ⟨expression⟩
```

where the keyword `A` denotes universal quantification and the keyword `E` denotes existential quantification. A quantified assignment can be put in parallel with other simple and quantified assignments, too. The range of quantification must be specified by constant integer expressions m and n where $m \leq n$.

The universal quantification behaves as expected. It is equivalent to a parallel assignment which contains the quantified assignment for all values of the quantified variable from the given range. Any of the partial assignments block the whole statement if it cannot be performed because of exceeded bounds. The universal quantification is very handy in operations with arrays. For instance, the following code initializes an array to zeros.

```
define ARRLNG 5
int arr[ARRLNG];
A i (0,ARRLNG-1) arr[i] = 0;
```

The existential quantification is more tricky and is currently beyond the scope of this document.

8.4 Break Statement

The syntax of a break statement is very simple. It involves only the keyword `break`.

```
break;
```

Outside a loop or a conditional, the statement does nothing. Within a clause of a loop or a conditional, it terminates the innermost loop or conditional and brings control to the lexically next statement after the loop or the conditional.

8.5 Conditionals and Loops

Conditional and loop statements of Remopla have much in common which is why this single section concerns them both. Both conditionals and loops consist of guarded clauses of statements where guards are boolean expressions (see section 7.2).

The syntax of a conditional statement is as follows. Guarded clauses belonging to the statement are delimited by the `if` and `fi` keywords of Remopla.

```
if
:: ⟨guard⟩ -> ⟨clause⟩;
...
:: ⟨guard⟩ -> ⟨clause⟩;
:: else   -> ⟨clause⟩;
fi
```

The syntax of a loop statement differs in the delimiting keywords only. They are `do` and `od` for a loop statement.

```
do
:: ⟨guard⟩ -> ⟨clause⟩;
...
:: ⟨guard⟩ -> ⟨clause⟩;
:: else   -> ⟨clause⟩;
od
```

Both conditionals and loops can contain at most one special keyword guard `else`.

A conditional functions as follows. When control arrives to a conditional, all guards are evaluated first. The one the clause of which gets control is then chosen non-deterministically among those guards that hold. It effectively means that the reachability analysis will consider each one of them. If no guard holds and there is an `else`-clause, control is passed to it. If there is neither a satisfied guard nor an `else`-clause, the execution aborts—no configuration is reachable from the one in which no clause can be chosen.

A loop functions in the same manner with the only exception that the control moves back to the beginning of the loop statement after execution of a clause.

The execution of both a loop and a conditional can be terminated with a `break` statement and a `goto` statement (see sections 8.4 and 8.2, respectively).

The semantics of loops and conditionals has several implications on resembling behaviour of loops and statements from common programming languages. Consider, for example, the following snippet of Remopla code.

```
if
:: a<b  -> a=b, b=a;
fi;
lbl: skip;
```

The label *lbl* is not reachable if `a>=b` which is likely not the intention in this case. The `else`-clause is required to fix it.

```
if
:: a<b  -> a=b, b=a;
:: else -> break;
fi;
lbl: skip;
```

Similarly, a Remopla implementation of a common while-loop can be achieved as follows.

```
do
:: while condition -> skip; # do something clever here
:: else -> break;
od;
```

Let us present one more example to demonstrates the non-deterministic choice among satisfied guards. The following `do`-statement will loop until execution of the last clause takes place. The value of `i` can then be any of `1`, `2` and `3`.

```
i = 1;
do
:: true -> i = 2;
:: true -> i = 3;
:: true -> break;
od;
```

9 Modules

Modules implement the concepts of functions and procedures in Remopla. A module can have arguments and can return a value. A module call can also be queried for reachability.

The Remopla language contains four distinct constructs concerning modules. An optional *module declaration* can be if a module is called before its definition in the source code of a model. A *module definition* just defines the behaviour of a module. A *module call* is used to pass control to a module. Finally, *return statement* returns control (and possibly a value) from a module to its caller.

9.1 Module Declaration

A module declaration declares a prototype of a module. I.e., it specifies the name (an identifier, see section 2), formal parameters and the type of return value of the module. The syntax of the declaration is as follows.

`module` ⟨return type⟩ ⟨module name⟩
`(`⟨parameter⟩`, ..., `⟨parameter⟩`);`

The return type can be one of

`void`
`bool` ⟨optional array specification⟩
`int` ⟨optional array specification⟩
`enum` ⟨enum-name⟩ ⟨optional array specification⟩
`struct` ⟨struct-name⟩ ⟨optional array specification⟩

where ⟨enum-name⟩ and ⟨struct-name⟩ must be names of globally declared enumerated and structured data types, respectively (see section 5). It is not possible to define a new data type in module declaration. Modules of `void` return type return no value.

The optional array specification is empty for simple return values. One- and two-dimensional arrays can also be specified. The syntax is as follows.

⟨data type⟩ `[`⟨m⟩`]`
⟨data type⟩ `[`⟨m⟩`][`⟨n⟩`]`

Formal parameters are given as a comma-separated list of single parameter declarations. The list can be empty. Each single parameter declaration has one of the following forms which are similar to variable declarations (see section 5). The only difference is that new data types cannot be defined in a declaration of a formal parameter. The identifier defines the name of the formal parameter being declared.

`bool` ⟨identifier⟩ ⟨optional array specification⟩
`int` ⟨identifier⟩ ⟨optional array specification⟩
`enum` ⟨enum-name⟩ ⟨identifier⟩ ⟨optional array specification⟩
`struct` ⟨struct-name⟩ ⟨identifier⟩ ⟨optional array specification⟩

9.2 Module Definition

Module definitions provide actual implementations of modules. They form a part of model body (see section 3). The structure of a module definition is as follows.

```
module ⟨return type⟩ ⟨module name⟩
          (⟨parameter⟩, ..., ⟨parameter⟩) {
     ⟨local declarations⟩
     ⟨statements⟩
}
```

The head has the same structure as module declaration (see above). In fact, the definition of a pre-declared module must exactly follow the declaration up to the terminating semicolon.

The body of a module definition starts with declarations of variables which are local to the module. The syntax of these declarations is exactly the same as described in section 5 with the only exception that it is not possible to define new data types here. A single name can be used for two distinct local variables of two modules. However, a local variable must not be of the same name as a global variable.

The rest of the definition body consists of a sequence of statements implementing the behaviour of the module. The statements include namely the `return` statement (see below) which returns control and a value to the caller of the module. Module definitions cannot be nested.

9.3 Return Statement

The return statement can only be used within a module and the form of the statement depends on the return type of the module. It aborts execution of the module and returns control to the caller of the module. It must also

specify the return value unless the module has void return type. The syntax of the statement is as follows.

```
return;
return ⟨return value⟩;
```

The type of the return value must be the same as return type of the module.

9.4 Module Call

Module calls are statements which pass control to a module along with the actual parameters the module requires. Modules can be called in two ways: as procedures and as functions.

A "procedure" module call, the syntax of which is as follows, discards any value the module might have returned.

```
⟨module name⟩ (⟨parameter⟩, ..., ⟨parameter⟩);
```

A "function" module call must be embedded in an assignment statement. The assigned variable must be of the same type as the return value of the module and the module call cannot be part of any expression for integer and boolean variables.

```
⟨variable identifier⟩ = ⟨module name⟩ (⟨parameters⟩);
```

The parameters are passed by value and must comply with the module definition. Values of integer and boolean parameters can be given as integer and boolean expressions, respectively.

The assignments of this type cannot be put in parallel with other assignments.

10 Remopla Syntax in BNF

10.1 Keywords

Keywords for types:
`bool`, `int`, `struct`, `enum`, `void`
Keywords for special values and constants:
`undef`, `false`, `true`, `DEFAULT_INT_BITS`
Keywords for statements:
`skip`, `if`, `fi`, `do`, `od`, `else`, `break`, `goto`, `return`
Other keywords:
`A`, `E`, `define`, `init`, `module`