# Model Syntax Reference for Package `isismdl`

2022-04-06

## Contents



# 1 Introduction

The model for package `isismdl` should be defined on an external ASCII file. This vignette describes the syntax of the Isis model file.

The `mdl` file defines a time-series model. The model language consists of one or more statements terminated by a semicolon (`;`). The language is case sensitive, in contrast to the model syntax for program Isis, which is case insensitive. The statements are prefixed by a statement identifier. The model language recognises the following statement identifiers : `param`, `frml`, `ident`, `function` and `end`.

# 2 Comments

The model definition file may contain comments. Comments start with `?` and extend to the end of line (inclusive). They may be placed anywhere in the model definition.

# 3 Identifiers

A legal identifier starts with a letter followed by zero or more letters, digits, underscore and `@`. Every identifier has a type

- variable
- parameter (`param` statement)
- user function (`function` statement)

All names may not be longer than 32 characters.

# 4 Variables

Variables can be used in expressions in the following ways

- `<varname>` indicating the current period value
- `<varname>[-<k>]` indicating a `<k>` period lag
- `<varname>[+<k>]` indicating a `<k>` period lead,

where `<varname>` is the name of the variable and `<k>` an unsigned integer constant. In this vignette, the words between `<` and `>` should not be interpreted literally: the word and the surrounding `<` and `>` signs should be substituted with a literal name or numerical constant. Examples of allowed expressions for variables are: `y` (current period value) and `y[-1]` (lag 1).

Variables occurring on the left hand side of an equation are endogenous. All other variables are exogenous (to the model).

For compatibility with older versions of `isismdl`, the lags and leads may also be specified with round parentheses `(-<k>)` instead of the square brackets `[-<k>]`.

# 5 Parameter statement

A `param` statement defines symbolic constants. The syntax is

```
param <parnameA> <valueA1>  [<valueA2> ...]
      [<parnameB> <valueB1> [<valueB2> ...] ]
      ;
```

where `[` and `]` delimit optional input. `<parnameA>` and `<parnameB>` are the names of the parameters, and `<valueA1>`, `<valueA2>` etc. are numerical constants.

When one value is specified for a parameter, the parameter is a scalar and can only be referenced by `<parname>`. When two or more values are specified, then the parameter is a vector. In that case, the first element is referenced by `<parname>`, and the other elements by `<parname>[-<k>]`, where `<k>` is an unsigned integer constant starting at 1 for the *second* element and ending at `N - 1` for the last element (`N` denotes the number of elements of the parameter).

For compatibility with older versions of `isismdl`, the elements of a parameter may also be specified with round parentheses `()` instead of the square brackets `[]`.

# 6 Equation statements

There are four types of equations. The syntax is

```
[ident] [<eqname>] <lhs var>    = <rhs expression>; # identity equation
[ident] [<eqname>] 0(<lhs var>) = <rhs expression>; # implicit identity equation
frml    [<eqname>] <lhs var>    = <rhs expression>; # stochastic equation
frml    [<eqname>] 0(<lhs var>) = <rhs expression>; # implicit stochastic equation,
```

where

- `<eqname>` (optional) the name of the equation (must be unique)
- `<lhs var>` the name of the left hand side variable
- `<rhs expression>` right hand side expression: a numerical expression or a single logical expression, which will be converted to numeric as if it was the argument to the built-in function `toreal`.

As before, `[` and `]` delimit optional input

All variables occurring on the left hand side of equations are endogenous variables. All others are exogenous.

The `ident` statement identifier is optional. The equation name is also optional. If not specified, then the equation will have the same name as the left hand side variable `<lhs var>`.

The syntax `0(<lhs var>)` defines an implicit equation, with `<lhs var>` as the implicit left hand side variable. It will be set to a value such that the right hand side (`<rhs expression>`) equals 0. `<lhs var>` must occur in `<rhs expression>`.

An equation of type `ident` (an equation with an `ident` statement identifier or equation without statement identifier) is an identity, i.e. the left hand side is determined exactly by the right hand side expression. An equation of type `frml` is a stochastic equation, i.e. the left hand side is determined from the right hand side expression `+` a residual. The residual is the so-called Constant Adjustment (or add factor) associated with the left hand side variable. The default value of a Constant Adjustment (CA) is 0.

A `frml` equation can be solved in two different modes.

1. The left hand side variable is endogenous and the CA is exogenous
2. The CA is endogenous and left hand side variable exogenous. The CA will be calculated as
   `<lhs var> - <rhs expression>`
   for an explicit `frml`, and as
   `0 - <rhs expression>`
   for an implicit `frml`, with the right hand side evaluated with the fixed value of the left hand side variable.

The default mode when solving the model is the first (endogenous left hand side variable).

# 7 Expressions

An expression consists of primary expressions and other expressions combined with operators. A primary expression (the basic building block) is

- constant
- parameter
- variable
- function call (built-in or user)

The type of an expression is either numerical or logical. Logical and numeric expressions may not be mixed. A logical expression used in a numeric context must be converted to numeric type through use of the built-in function `toreal`.

## 7.1 Operators

The operators recognised in expressions are

| Operator | Meaning |
|---|---|
| `.or.` or `\|` | logical OR |
| `.and.` or `&` | logical AND |
| `.not.` or `^` | logical NOT |

| Operator | Meaning |
|---|---|
| `= ^= > >= < <=` | relational operators |
| `+ -` | addition / subtraction |
| `* /` | multiplication / division |
| `- +` | unary minus / unary plus |
| `**` | exponentiation |
| `( )` | grouping |

The operators are listed in order of increasing precedence. Operators with a higher precedence are evaluated before operators of a lower precedence. All operators are evaluated from left to right, with exponentiation as the exception (right to left). This implies that

```
a ** b ** 2
```

is evaluated as

```
 a ** (b ** 2)
```

The grouping operators `(` and `)` let you override the normal operator precedence.

The logical or boolean operators, `.and.`, `.or.`, and `.not.`, are used in logical expressions. Logical expressions are always evaluated completely: there is no short circuited boolean evaluation. They can only be used on logical values: true or false.

The relational operators, `=`, `^=`, `>=`, `>`, `<=`, and `<`, compare their left hand side with their right hand side and evaluate to true or false. These operators can only be used on numerical operands. They are not associative (`a > b > c` is illegal).

## 7.2 Built-in functions

Currently available built-in functions are listed in the following table

| Function | Meaning |
|---|---|
| `log` | natural logarithm |
| `log10` | base 10 logarithm |
| `exp` | exponential |
| `sin` | sine |
| `cos` | cosine |
| `tan` | tangent |
| `asin` | arcsine |
| `acos` | arc cosine |
| `atan` | arctangent |
| `sinh` | hyperbolic sine |
| `cosh` | hyperbolic cosine |
| `tanh` | hyperbolic tangent |
| `abs` | absolute value |
| `sqrt` | square root |
| `nint` | round to nearest integer |
| `max` | maximum |
| `min` | minimum |
| `toreal` | convert logical value to numerical value |
| `hypot` | $\text{hypot}(x, y) = \sqrt{x^2 + y^2}$ |
| `fibur` | Fischer-Burmeister function: $\text{fibur}(x, y) = \sqrt{x^2 + y^2} - (x + y)$ |

All functions take one expression as argument with the following exceptions:

- `max` and `min` take at least two expressions as arguments
- `hypot` and `fibur` take two expressions as arguments.

The function `toreal` converts a `true` argument to 1 and a `false` argument to 0.

## 7.3 `if` expression

The syntax of an `if` expression is

```
if <condition_expr> then
   <evaluate_expr_if>
[elseif <elseif_condition_expr_1> then
   <evaluate_expr_elseif_1>]
...
else
   <evaluate_expr_else>
endif
```

where the condition expressions (`<condition_expr>` and `elseif_condition_expr_1>`) are logical expressions and the evaluate expressions (`<evaluate_expr_if>`, `<evaluate_expr_elseif_1>` and `<evaluate_expr_else>`) numerical or logical expressions. The result of the `if` expression is the same as that of the evaluated expression. All evaluate expressions must have the same type (numerical or logical).

The `else` part is required, while the `elseif` part is optional. There can be any number of `elseif` parts in an `if` expression. `if` expressions may be nested. An `if` expression can be used in the `condition` of another `if` expression.

In older versions of the Isis model syntax there was no `endif` keyword: the end of the `<evaluate_expr_else>` was implicit. Because the current version of `isismdl` should compile all models that could be compiled with older versions, the `endif` keyword may still be omitted (not recommended). If the `endif` is omitted, then the end of the `<evaluate_else_expr>` is implicit: it can be the end of the current statement (`;`), the end of the current grouped expression (`)`), or the end of the current function argument (`,`). This implies that

```
<expr> + if <condition> then <if_expr> else <else_expr>
```

is not the same as

```
if <condition> then <if_expr> else <else_expr> + <expr>
```

In the first case, the result is `<expr> + <if_expr>` or `<expr> + <else_expr>`, whereas in the second case it is either `<if_expr>` or `<else_expr> + <expr>`. Use the `endif` keyword to force the correct order of evaluation as in

```
if  <condition> then <if_expr> else <else_expr>  endif + <expr>
```

## 7.4 `sum` function

The `sum` function provides summation of expressions. The syntax is

```
sum(<index> = <lo>, <hi> : <expr>)
```

where

- `<index>` the name of the index for the summation
- `<lo>` lower bound of the sum index, specified as an (un)signed integer
- `<hi>` upper bound of the sum index, specified as an (un)signed integer
- `<expr>` numeric expression to sum

`sum` functions may not be nested.

Within <expr>, the summation index <index> can be used as any other variable. Within the scope of the sum function, a model variable or parameter with the same name as index is inaccessible. In <expr>, lags and leads may depend on the <index> in the following ways

- <name>[<index>]
- <name>[<index> + <integer>]
- <name>[<index> - <integer>]

where <name> refers to a parameter or variable. For compatibility with older versions of isismdl, the lag or lead of a variable or parameters may also be specified with round parentheses () instead of the square brackets [].

### 7.5 del function

The del function provides first order differencing of expressions. The syntax is

del(<delnum> : <expr>)

where

- <delnum> unsigned integer giving the lag for which to calculate the first difference
- <expr> numeric expression to which to apply first order differencing.

del functions may not be nested. First order differencing is only applied to variables, not to parameters.

## 8 User functions

A user function is defined as follows

function <funcname>(<arg1>, ... <argN>) = <expr>;

where

- <funcname> the name of the function
- <arg1> the name of the first formal argument
- <argN> the name of the last formal argument
- <expr> an expression, which may refer to formal arguments, model variables and model parameters.

User functions must be defined before they are used. User functions may call other user functions that have been defined before, as well as all built-in functions. Expressions passed as arguments may be logical or numeric. If an argument is of type logical, it can only be used directly in the condition part of an if expression or the argument of the built-in function toreal.

<expr> may apply lags or leads to function arguments. In that case a function argument may only be a simple variable followed (optionally) by a lag or lead indicator. A function may return a logical value.

## 9 End statement

The syntax is

end;

All input after this statement is ignored. It is optional. The end statement is not allowed in files included with the #include preprocessor directive (see Section 10.1). Instead of the end statement, it is possible to use conditional compilation using a dummy flag (see 10.2.1).

# 10 The model preprocessor

The model compiler of `isismdl` provides a simple preprocessor. The preprocessor is a first step in the model compilation in which the input file is processed and the input for the actual model compilation is generated. The preprocessor commands all start with the `#` sign. For example, `#include` is used to include the contents of a file during model compilation. Another feature is conditional compilation using the `#if`, `#else`, `#elseif` and `#endif` directives (see Section 10.2.

Unfortunately, model analysis program Nephthys cannot handle preprocessor directives yet, so do not use the model preprocessor if you want to use Nephthys.

## 10.1 File inclusion

The `#include` directive can be used to insert functions, parameters and model equations from another input file. The syntax is

```
#include "<filename>"
```

where `<filename>` is the name of the file to include.

`<filename>` can be an absolute or relative path of a file. If it is a relative path, then the model compiler searches for the file in the same directory as where the source file is located (the source file is the file with the `#include` directive). If not found there, it searches in the current directory. It is possible to add directories to the search path in function `isis_mdl`.

It is not allowed to use the end statement (see Section 9) in an included file. If you wish to skip some equations at the end of the file, use conditional compilation (see section 10.2.1) with a dummy flag.

## 10.2 Conditional compilation

Sometimes it is useful to include or skip equations or parts of equations in the model compilation. Conditional compilation makes this possible.

The general syntax of the preprocessor conditional statement is:

```
#if <flag1>
  <code1>
[#elseif <flag2>
  <code2>]
...
[# else
  <codeN>]
#endif
```

`<flag1>` and `<flag2>` are so called compiler flags that can be specified with argument `parse_options` of function `isis_mdl`. The `#if` directive tests whether flag `<flag1>` has been specified. If it has been specified, then the model code `<code1>` is compiled. Otherwise the code `<code1>` is ignored. The `#elseif` and `#else` parts are optional. A simple example of a model with an `#if` directive:

```
#if x_zero
    x = 0;
#endif
y = x;
```

If we use the following R code

```
library(isismdl)
isis_mdl("example.mdl", parse_options = list(flags = "x_zero"))
```

```
## Isis Model Compiler 3.00
## Compiling model ...
##        2 equations processed
## Ordering equations ...
## Writing MIF file ...
## Writing cross-reference file ...
## End compilation
## Reading mif file ...
## Model with     2 equations read
## Checking Model-code ...
## Model is ok ...
## No feedback ordering for this model ...
```

```
## IsisMdl object
## Model index:                                          1
## Number of variables:                                  2
## Maximum lag:                                          0
## Maximum lead:                                         0
```

then equation `x = 0;` is included. However, if the compiler flag `x_zero` is not specified, as in the example below, then the equation `x = 0;` is not included.

```
isis_mdl("example.mdl")
```

```
## Isis Model Compiler 3.00
## Compiling model ...
##        1 equations processed
## Ordering equations ...
## Writing MIF file ...
## Writing cross-reference file ...
## End compilation
## Reading mif file ...
## Model with     1 equations read
## Checking Model-code ...
## Model is ok ...
## No feedback ordering for this model ...
```

```
## IsisMdl object
## Model index:                                          2
## Number of variables:                                  2
## Maximum lag:                                          0
## Maximum lead:                                         0
```

The code used in the preprocessor conditional statements can be any model code. It may also be a part of an equation, as in the following examples

```
ident x = #if x_zero 0 #else 1 #endif
```

### 10.2.1 Using conditional compilation to skip equations

Conditional compilation can also be used to skip some equations during compilation. For example

```
#if dummy
x = 1;
z = 2;
#endif
```

If the compiler flag `dummy` has not been defined, the equations `x = 1` and `z = 2` are skipped.