九州大学
KYUSHU UNIVERSITY

VDMTools

The VDM-SL Language Manual

VDM

How to contact:

http://fmvdm.org/	VDM information web site(in Japanese)
http://fmvdm.org/tools/vdmtools	VDMTools web site(in Japanese)
inq@fmvdm.org	Mail

The VDM-SL Language Manual 2.0
— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice.

Contents

1 Introduction

This document describes the syntax and semantics of the VDM-SL language which is essentially standard ISO/VDM-SL [5] with a modular extension [1]. Notice that all syntactically correct VDM-SL specifications are also correct in VDM-SL. Even though we have tried to present the language in a clear and understandable way the document is not a complete VDM-SL reference manual. For a more thorough presentation of the language we refer to the existing literature[2]. Wherever the VDM-SL notation differs from the VDM-SL standard notation the semantics will of course be carefully explained.

The VDM-SL language is the language supported by the VDM-SL Toolbox (see [?]). This Toolbox contains a syntax checker, a static semantics checker, an interpreter[3] and a code generator to C++. Because ISO/VDM-SL in general is a non-executable language the interpreter supports only a subset of the language. This document will focus particularly on the points where the semantics of VDM-SL differs from the semantics used in the interpreter. In this document we will use the term "the interpreter" whenever we refer to the interpreter from the VDM-SL Toolbox, and we will refer to "VDM-SL" whenever the semantics of some language construct is totally identical to the dynamic semantics for the VDM-SL standard.

Consequently we will use the ASCII (also called the interchange) concrete syntax but we will display all reserved words in a special keyword font. This is done because the document works as a language manual to the VDM-SL Toolbox where the ASCII notation is used as input. The mathematical concrete syntax can be generated automatically by the Toolbox so a nicer looking syntax can be produced.

Section 2 indicates how the language presented here and the corresponding VDM-SL Toolbox conform to the VDM-SL standard. Section 3 presents the BNF notation used for the description of syntactic constructs. The VDM-SL notation is described in section 4 to section 14. Section 16 provides a complete list of the differences between ISO/VDM-SL and VDM-SL while section 17 contains a short explanation of the static semantics of VDM-SL. The complete syntax of the language is described in Appendix A, the lexical specification in Appendix B and the operator precedence in Appendix C. Appendix D presents a list of the

[1] A few other extensions are also included.

[2] A more tutorial like presentation is given in [3] whereas proofs in VDM-SL are treated best in [4] and [1].

[3] In addition the Toolbox provides pretty printing facilities, debugging facilities and support for test coverage, but these are the basic components.

differences between symbols in the mathematical syntax and the ASCII concrete syntax. In Appendex E details of the Standard library and how to use it are given. Finally, an index of the defining occurrences of all the syntax rules in the document is given.

2 Conformance Issues

The VDM-SL standard has a conformance clause which specifies a number of levels of conformity. The lowest level of conformity deals with syntax conformance. The VDM-SL Toolbox accepts specifications which follow the syntax description in the standard.

In addition it accepts a number of extensions (see section 16) which should be rejected according to the conformance clause.

Level one in the conformance clause deals with the static semantics for possible correctness (see section 17). In this part we have chosen to reject more specifications than the standard prescribes as being possibly well-formed[4].

Level two and the following levels (except the last one) deal with the definite well-formedness static semantics check and a number of possible extended checks which can be added to the static semantics. The definitely well-formedness check is present in the Toolbox. However, we do not consider it to be of major value for real examples because almost no "real" specifications will be able to pass this test.

The last conformance level deals with the dynamic semantics. Here it is required that an accompanying document provides details about the deviations from the standard dynamic semantics (which is not executable). This is actually done in this document by explaining which constructs can be interpreted by the Toolbox and what the deviations are for a few constructs. Thus, this level of conformance is satisfied by the VDM-SL Toolbox.

To sum up, we can say that VDM-SL (and its supporting Toolbox) is quite close conforming to the standard, but we have not yet invested the time in ensuring this.

[4]For example with a set comprehension where a predicate is present the standard does not check the element expression at all (in the possibly well-formedness check) because the predicate could yield false (and thus the whole expression would just be another way to write an empty set). We believe that a user will be interested in getting such parts tested as well.

3 Concrete Syntax Notation

Wherever the syntax for parts of the language is presented in the document it will be described in a BNF dialect. The BNF notation used employs the following special symbols:

,	the concatenate symbol
=	the define symbol
|	the definition separator symbol (alternatives)
[ ]	enclose optional syntactic items
{ }	enclose syntactic items which may occur zero or more times
‘ ’	single quotes are used to enclose terminal symbols
meta identifier	non-terminal symbols are written in lower-case letters (possibly including spaces)
;	terminator symbol to denote the end of a rule
( )	used for grouping, e.g. “a, (b | c)” is equivalent to “a, b | a, c”.
–	denotes subtraction from a set of terminal symbols (e.g. “character – (‘"’)” denotes all characters excepting the double quote character.)

4 Data Type Definitions

As in traditional programming languages it is possible to define data types in VDM-SL and give them appropriate names. Such an equation might look like:

```
Amount = nat
```

Here we have defined a data type with the name “`Amount`” and stated that the values which belong to this type are natural numbers (`nat` is one of the basic types described below). One general point about the type system of VDM-SL which is worth mentioning at this point is that equality and inequality can be used between any value. In programming languages it is often required that the operands have the same type. Because of a construct called a union type (described below) this is not the case for VDM-SL.

In this section we will present the syntax of data type definitions. In addition, we will show how values belonging to a type can be constructed and manipulated (by means of built-in operators). We will present the basic data types first and then we will proceed with the compound types.

4.1 Basic Data Types

In the following a number of basic types will be presented. Each of them will contain:

- Name of the construct.
- Symbol for the construct.
- Special values belonging to the data type.
- Built-in operators for values belonging to the type.
- Semantics of the built-in operators.
- Examples illustrating how the built-in operators can be used.[5]

For each of the built-in operators the name, the symbol used and the type of the operator will be given together with a description of its semantics (except that the semantics of Equality and Inequality is not described, since it follows the usual semantics). In the semantics description identifiers refer to those used in the corresponding definition of operator type, e.g. `a`, `b`, `x`, `y` etc.

The basic types are the types defined by the language with distinct values that cannot be analysed into simpler values. There are five fundamental basic types: booleans, numeric types, characters, tokens and quote types. The basic types will be explained one by one in the following.

4.1.1 The Boolean Type

In general VDM-SL allows one to specify systems in which computations may fail to terminate or to deliver a result. To deal with such potential undefinedness, VDM-SL employs a three valued logic: values may be true, false or bottom (undefined). The semantics of the interpreter differs from VDM-SL in that it

[5]In these examples the Meta symbol '≡' will be used to indicate what the given example is equivalent to.

does not have an LPF (Logic of Partial Functions) three valued logic where the order of the operands is unimportant (see [4]). The and operator, the or operator and the imply operator, though, have a conditional semantics meaning that if the first operand is sufficient to determine the final result, the second operand will not be evaluated. In a sense the semantics of the logic in the interpreter can still be considered to be three-valued as for VDM-SL. However, bottom values may either result in infinite computation or a run-time error in the interpreter.

Name: Boolean

Symbol: bool

Values: true, false

Operators: Assume that a and b in the following denote arbitrary boolean expressions:

Operator	Name	Type
`not b`	Negation	bool $\rightarrow$ bool
`a and b`	Conjunction	bool $*$ bool $\rightarrow$ bool
`a or b`	Disjunction	bool $*$ bool $\rightarrow$ bool
`a => b`	Implication	bool $*$ bool $\rightarrow$ bool
`a <=> b`	Biimplication	bool $*$ bool $\rightarrow$ bool
`a = b`	Equality	bool $*$ bool $\rightarrow$ bool
`a <> b`	Inequality	bool $*$ bool $\rightarrow$ bool

Semantics of Operators: Semantically <=> and = are equivalent when we deal with boolean values. There is a conditional semantics for and, or and =>.

We denote undefined terms (e.g. applying a map with a key outside its domain) by $\perp$. The truth tables for the boolean operators are then[6]:

[6] Notice that in standard VDM-SL all these truth tables (except =>) would be symmetric.

Negation `not b`

b	true	false	$\perp$
not b	false	true	$\perp$

Conjunction `a and b`

$a\backslash b$	true	false	$\perp$
true	true	false	$\perp$
false	false	false	false
$\perp$	$\perp$	$\perp$	$\perp$

Disjunction `a or b`

$a\backslash b$	true	false	$\perp$
true	true	true	true
false	true	false	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Implication `a => b`

$a\backslash b$	true	false	$\perp$
true	true	false	$\perp$
false	true	true	true
$\perp$	$\perp$	$\perp$	$\perp$

Biimplication `a <=> b`

$a\backslash b$	true	false	$\perp$
true	true	false	$\perp$
false	false	true	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Examples: Let `a = true` and `b = false` then:

not a	$\equiv$	false
a and b	$\equiv$	false
b and $\perp$	$\equiv$	false
a or b	$\equiv$	true
a or $\perp$	$\equiv$	true
a => b	$\equiv$	false
b => b	$\equiv$	true
b => $\perp$	$\equiv$	true
a <=> b	$\equiv$	false
a = b	$\equiv$	false
a <> b	$\equiv$	true
$\perp$ or not $\perp$	$\equiv$	$\perp$
(b and $\perp$) or ($\perp$ and false)	$\equiv$	$\perp$

4.1.2 The Numeric Types

There are five basic numeric types: positive naturals, naturals, integers, rationals and reals. Except for three, all the numerical operators can have mixed operands of the three types. The exceptions are integer division, modulo and the remainder operation.

The five numeric types denote a hierarchy where real is the most general type followed by rat[7], int, nat and nat1.

Type	Values
nat1	1, 2, 3, ...
nat	0, 1, 2, ...
int	..., -2, -1, 0, 1, ...
real	..., -12.78356, ..., 0, ..., 3, ..., 1726.34, ...

This means that any number of type int is also automatically of type real but not necessarily of type nat. Another way to illustrate this is to say that the positive natural numbers are a subset of the natural numbers which again are a subset of the integers which again are a subset of the rational numbers which finally are a subset of the real numbers. The following table shows some numbers and their associated type:

Number	Type
3	real, rat, int, nat, nat1
3.0	real, rat, int, nat, nat1
0	real, rat, int, nat
-1	real, rat, int
3.1415	real, rat

Note that all numbers are necessarily of type real (and rat).

Names: real, rational, integer, natural and positive natural numbers.

Symbols: real, rat, int, nat, nat1

Values: ..., -3.89, ..., -2, ..., 0, ..., 4, ..., 1074.345, ...

Operators: Assume in the following that x and y denote numeric expressions. No assumptions are made regarding their type.

[7] From the VDM-SL Toolbox's point of view there is no difference between real and rat because only rational numbers can be represented in a computer.

Operator	Name	Type
`-x`	Unary minus	real → real
`abs x`	Absolute value	real → real
`floor x`	Floor	real → int
`x + y`	Sum	real ∗ real → real
`x - y`	Difference	real ∗ real → real
`x * y`	Product	real ∗ real → real
`x / y`	Division	real ∗ real → real
`x div y`	Integer division	int ∗ int → int
`x rem y`	Remainder	int ∗ int → int
`x mod y`	Modulus	int ∗ int → int
`x**y`	Power	real ∗ real → real
`x < y`	Less than	real ∗ real → bool
`x > y`	Greater than	real ∗ real → bool
`x <= y`	Less or equal	real ∗ real → bool
`x >= y`	Greater or equal	real ∗ real → bool
`x = y`	Equal	real ∗ real → bool
`x <> y`	Not equal	real ∗ real → bool

The types stated for operands are the most general types allowed. This means for instance that unary minus works for operands of all five types (nat1, nat, int rat and real).

Semantics of Operators: The operators Unary minus, Sum, Difference, Product, Division, Less than, Greater than, Less or equal, Greater or equal, Equal and Not equal have the usual semantics of such operators.

Operator Name	Semantics Description
Floor	yields the greatest integer which is equal to or smaller than `x`.
Absolute value	yields the absolute value of `x`, i.e. `x` itself if `x >= 0` and `-x` if `x < 0`.
Power	yields `x` raised to the `y`'th power.

There is often confusion on how integer division, remainder and modulus work on negative numbers. In fact, there are two valid answers to `-14 div 3`: either (the intuitive) `-4` as in the Toolbox, or `-5` as in e.g. Standard ML [6]. It is therefore appropriate to explain these operations in some detail.

Integer division is defined using floor and real number division:

```
x/y <  0:    x div y = -floor(abs(-x/y))
```

```
x/y >= 0:   x div y =  floor(abs(x/y))
```

Note that the order of floor and abs on the right-hand side makes a difference, the above example would yield -5 if we changed the order. This is because floor always yields a smaller (or equal) integer, e.g. floor (14/3) is 4 while floor (-14/3) is -5.

Remainder x rem y and modulus x mod y are the same if the signs of x and y are the same, otherwise they differ and rem takes the sign of x and mod takes the sign of y. The formulas for remainder and modulus are:

```
x rem y = x - y * (x div y)
x mod y = x - y * floor(x/y)
```

Hence, -14 rem 3 equals -2 and -14 mod 3 equals 1. One can view these results by walking the real axis, starting at -14 and making jumps of 3. The remainder will be the last negative number one visits, because the first argument corresponding to x is negative, while the modulus will be the first positive number one visit, because the second argument corresponding to y is positive.

Examples: Let a = 7, b = 3.5, c = 3.1415, d = -3, e = 2 then:

- a	≡	-7
abs a	≡	7
abs d	≡	3
floor a <= a	≡	true
a + d	≡	4
a * b	≡	24.5
a / b	≡	2
a div e	≡	3
a div d	≡	-2
a mod e	≡	1
a mod d	≡	-2
-a mod d	≡	-1
a rem e	≡	1
a rem d	≡	1
-a rem d	≡	-1
3**2 + 4**2 = 5**2	≡	true
b < c	≡	false
b > c	≡	true

`a <= d`	$\equiv$	false
`b >= e`	$\equiv$	true
`a = e`	$\equiv$	false
`a = 7.0`	$\equiv$	true
`c <> d`	$\equiv$	true
`abs c < 0`	$\equiv$	false
`(a div e) * e`	$\equiv$	6

4.1.3 The Character Type

The character type contains all the single character elements of the VDM character set (see Table 11 on page 145).

Name: Char

Symbol: char

Values: `'a', 'b', ..., '1', '2', ...'+', '-' ...`

Operators: Assume that `c1` and `c2` in the following denote arbitrary characters:

Operator	Name	Type
`c1 = c2`	Equal	char $*$ char $\rightarrow$ bool
`c1 <> c2`	Not equal	char $*$ char $\rightarrow$ bool

Examples:

`'a' = 'b'`	$\equiv$	false
`'1' = 'c'`	$\equiv$	false
`'d' <> '7'`	$\equiv$	true
`'e' = 'e'`	$\equiv$	true

4.1.4 The Quote Type

The quote type corresponds to enumerated types in a programming language like Pascal. However, instead of writing the different quote literals between curly brackets in VDM-SL it is done by letting a quote type consist of a single quote literal and then let them be a part of a union type.

Name: Quote

Symbol: e.g. <QuoteLit>

Values: <RED>, <CAR>, <QuoteLit>, ...

Operators: Assume that q and r in the following denote arbitrary quote values belonging to an enumerated type T:

Operator	Name	Type
q = r	Equal	$T * T \to$ bool
q <> r	Not equal	$T * T \to$ bool

Examples: Let T be the type defined as:

```
T = <France> | <Denmark> | <SouthAfrica> | <SaudiArabia>
```

If for example a = <France> then:

<France> = <Denmark>	$\equiv$	false
<SaudiArabia> <> <SouthAfrica>	$\equiv$	true
a <> <France>	$\equiv$	false

4.1.5 The Token Type

The token type consists of a countably infinite set of distinct values, called tokens. The only operations that can be carried out on tokens are equality and inequality. In VDM-SL, tokens cannot be individually represented whereas they can be written with a mk_token around an arbitrary expression. This is a way of enabling testing of specifications which contain token types. However, in order to resemble the VDM-SL standard these token values cannot be decomposed by means of any pattern matching and they cannot be used for anything other than equality and inequality comparisons.

Name: Token

Symbol: token

Values: mk_token(5), mk_token({9, 3}), mk_token([true, {}]), ...

Operators: Assume that s and t in the following denote arbitrary token values:

Operator	Name	Type
s = t	Equal	token $*$ token $\to$ bool
s <> t	Not equal	token $*$ token $\to$ bool

Examples: Let for example `s = mk_token(6)` and let `t = mk_token(1)` in:

`s = t`	≡	false
`s <> t`	≡	true
`s = mk_token(6)`	≡	true

4.2 Compound Types

In the following compound types will be presented. Each of them will contain:

- The syntax for the compound type definition.
- An equation illustrating how to use the construct.
- Examples of how to construct values belonging to the type. In most cases there will also be given a forward reference to the section where the syntax of the basic constructor expressions is given.
- Built-in operators for values belonging to the type [8].
- Semantics of the built-in operators.
- Examples illustrating how the built-in operators can be used.

For each of the built-in operators the name, the symbol used and the type of the operator will be given together with a description of its semantics (except that the semantics of Equality and Inequality is not described, since it follows the usual semantics). In the semantics description identifiers refer to those used in the corresponding definition of operator type, e.g. `m`, `m1`, `s`, `s1` etc.

4.2.1 Set Types

A set is an unordered collection of values, all of the same type[9], which is treated as a whole. All sets in VDM-SL are finite, i.e. they contain only a finite number of elements. The elements of a set type can be arbitrarily complex, they could for example be sets themselves.

[8] These operators are used in either unary or binary expressions which are given with all the operators in section 7.3.

[9] Note however that it is always possible to find a common type for two values by the use of a union type (see section 4.2.6.)

In the following this convention will be used: `A` is an arbitrary type, `S` is a set type, `s`, `s1`, `s2` are set values, `ss` is a set of set values, `e`, `e1`, `e2` and `en` are elements from the sets, `bd1, bd2`, ..., `bdm` are bindings of identifiers to sets or types, and `P` is a logical predicate.

Syntax: type = set type
| ... ;

set type = 'set of', type ;

Equation: `S = set of A`

Constructors:

Set enumeration: `{e1, e2, ..., en}` constructs a set of the enumerated elements. The empty set is denoted by {}.

Set comprehension: `{e | bd1, bd2, ..., bdm & P}` constructs a set by evaluating the expression `e` on all the bindings for which the predicate `P` evaluates to `true`. A binding is either a set binding or a type binding[10]. A set bind `bdn` has the form `pat1, ..., patp in set s`, where `pati` is a pattern (normally simply an identifier), and `s` is a set constructed by an expression. A type binding is similar, in the sense that `in set` is replaced by a colon and `s` is replaced with a type expression.

The syntax and semantics for all set expressions are given in section 7.7.

Operators:

[10]Notice that type bindings cannot be executed by the interpreter because in general they are not executable (see section 9 for further information about this).

Operator	Name	Type
`e in set s1`	Membership	$A *$ set of $A \to$ bool
`e not in set s1`	Not membership	$A *$ set of $A \to$ bool
`s1 union s2`	Union	set of $A *$ set of $A \to$ set of A
`s1 inter s2`	Intersection	set of $A *$ set of $A \to$ set of A
`s1 \ s2`	Difference	set of $A *$ set of $A \to$ set of A
`s1 subset s2`	Subset	set of $A *$ set of $A \to$ bool
`s1 psubset s2`	Proper subset	set of $A *$ set of $A \to$ bool
`s1 = s2`	Equality	set of $A *$ set of $A \to$ bool
`s1 <> s2`	Inequality	set of $A *$ set of $A \to$ bool
`card s1`	Cardinality	set of $A \to$ nat
`dunion ss`	Distributed union	set of set of $A \to$ set of A
`dinter ss`	Distributed intersection	set of set of $A \to$ set of A
`power s1`	Finite power set	set of $A \to$ set of set of A

Note that the types `A`, `set of A` and `set of set of A` are only meant to illustrate the structure of the type. For instance it is possible to make a union between two arbitrary sets `s1` and `s2` and the type of the resultant set is the union type of the two set types. Examples of this will be given in section 4.2.6.

Semantics of Operators:

Operator Name	Semantics Description
Membership	tests if `e` is a member of the set `s1`
Not membership	tests if `e` is not a member of the set `s1`
Union	yields the union of the sets `s1` and `s2`, i.e. the set containing all the elements of both `s1` and `s2`.
Intersection	yields the intersection of sets `s1` and `s2`, i.e. the set containing the elements that are in both `s1` and `s2`.
Difference	yields the set containing all the elements from `s1` that are not in `s2`. `s2` need not be a subset of `s1`.
Subset	tests if `s1` is a subset of `s2`, i.e. whether all elements from `s1` are also in `s2`. Notice that any set is a subset of itself.
Proper subset	tests if `s1` is a proper subset of `s2`, i.e. it is a subset and `s2\s1` is non-empty.
Cardinality	yields the number of elements in `s1`.
Distributed union	the resulting set is the union of all the elements (these are sets themselves) of `ss`, i.e. it contains all the elements of all the elements/sets of `ss`.

Operator Name	Semantics Description
Distributes intersection	the resulting set is the intersection of all the elements (these are sets themselves) of, i.e. it contains the elements that are in all the elements/sets of `ss`. `ss` must be non-empty.
Finite power set	yields the power set of `s1`, i.e. the set of all subsets of `s1`.

Examples: Let `s1 = {<France>,<Denmark>,<SouthAfrica>,<SaudiArabia>}`, `s2 = {2, 4, 6, 8, 11}` and `s3 = {}` then:

```
<England> in set s1                       ≡ false
10 not in set s2                          ≡ true
s2 union s3                               ≡ {2, 4, 6, 8, 11}
s1 inter s3                               ≡ {}
(s2 \ {2,4,8,10}) union {2,4,8,10} = s2   ≡ false
s1 subset s3                              ≡ false
s3 subset s1                              ≡ true
s2 psubset s2                             ≡ false
s2 <> s2 union {2, 4}                     ≡ false
card s2 union {2, 4}                      ≡ 5
dunion {s2, {2,4}, {4,5,6}, {0,12}}       ≡ {0,2,4,5,6,8,11,12}
dinter {s2, {2,4}, {4,5,6}}               ≡ {4}
dunion power {2,4}                        ≡ {2,4}
dinter power {2,4}                        ≡ {}
```

4.2.2 Sequence Types

A sequence value is an ordered collection of elements of some type indexed by `1, 2, ..., n`; where `n` is the length of the sequence. A sequence type is the type of finite sequences of elements of a type, either including the empty sequence (seq0 type) or excluding it (seq1 type). The elements of a sequence type can be arbitrarily complex; they could e.g. be sequences themselves.

In the following this convention will be used: `A` is an arbitrary type, `L` is a sequence type, `S` is a set type, `l`, `l1`, `l2` are sequence values, `ll` is a sequence of sequence values. `e1`, `e2` and `en` are elements in these sequences, `i` will be a natural number, `P` is a predicate and `e` is an arbitrary expression.

Syntax: type = seq type

| ... ;

seq type = seq0 type
| seq1 type ;

seq0 type = ‘seq of’, type ;

seq1 type = ‘seq1 of’, type ;

Equation: `L = seq of A` or `L = seq1 of A`

Constructors:

Sequence enumeration: `[e1, e2,..., en]` constructs a sequence of the enumerated elements. The empty sequence will be written as `[]`. A text literal is a shorthand for enumerating a sequence of characters (e.g. `"ifad" = ['i','f','a','d']`).

Sequence comprehension: `[e | id in set S & P]` constructs a sequence by evaluating the expression `e` on all the bindings for which the predicate `P` evaluates to `true`. The expression `e` will use the identifier `id`. `S` is a set of numbers and `id` will be matched to the numbers in the normal order (the smallest number first).

The syntax and semantics of all sequence expressions are given in section 7.8.

Operators:

Operator	Name	Type
`hd l`	Head	seq1 of $A \rightarrow A$
`tl l`	Tail	seq1 of $A \rightarrow$ seq of A
`len l`	Length	seq of $A \rightarrow$ nat
`elems l`	Elements	seq of $A \rightarrow$ set of A
`inds l`	Indexes	seq of $A \rightarrow$ set of nat1
`l1 ^ l2`	Concatenation	(seq of A) $*$ (seq of A) $\rightarrow$ seq of A
`conc ll`	Distributed concatenation	seq of seq of $A \rightarrow$ seq of A
`l ++ m`	Sequence modification	seq of $A *$ map nat1 to $A \rightarrow$ seq of A
`l(i)`	Sequence application	seq of $A *$ nat1 $\rightarrow A$
`l1 = l2`	Equality	(seq of A) $*$ (seq of A) $\rightarrow$ bool
`l1 <> l2`	Inequality	(seq of A) $*$ (seq of A) $\rightarrow$ bool

The type `A` is an arbitrary type and the operands for the concatenation and distributed concatenation operators do not have to be of the same (`A`) type. The type of the resultant sequence will be the union type of the types of the operands. Examples will be given in section 4.2.6.

Semantics of Operators:

Operator Name	Semantics Description
Head	yields the first element of `l`. `l` must be a non-empty sequence.
Tail	yields the subsequence of `l` where the first element is removed. `l` must be a non-empty sequence.
Length	yields the length of `l`.
Elements	yields the set containing all the elements of `l`.
Indexes	yields the set of indexes of `l`, i.e. the set {`1,...,len l`}.
Concatenation	yields the concatenation of `l1` and `l2`, i.e. the sequence consisting of the elements of `l1` followed by those of `l2`, in order.
Distributed concatenation	yields the sequence where the elements (these are sequences themselves) of `ll` are concatenated: the first and the second, and then the third, etc.
Sequence modification	the elements of `l` whose indexes are in the domain of `m` are modified to the range value that the index maps into. `dom m` must be a subset of inds `l`
Sequence application	yields the element of index from `l`. `i` must be in the indexes of `l`.

Examples: Let `l1 = [3,1,4,1,5,9,2]`, `l2 = [2,7,1,8]`,
`l3 = [<England>, <Rumania>, <Colombia>, <Tunisia>]` then:

```
len l1                                  ≡ 7
hd (l1^l2)                              ≡ 3
tl (l1^l2)                              ≡ [1,4,1,5,9,2,2,7,1,8]
l3(len l3)                              ≡ <Tunisia>
"England"(2)                            ≡ 'n'
conc [l1,l2] = l1^l2                    ≡ true
conc [l1,l1,l2] = l1^l2                 ≡ false
elems l3                                ≡ { <England>, <Rumania>,
                                            <Colombia>,<Tunisia>}
(elems l1) inter (elems l2)             ≡ {1,2}
inds l1                                 ≡ {1,2,3,4,5,6,7}
(inds l1) inter (inds l2)               ≡ {1,2,3,4}
l3 ++ {2 |-> <Germany>,4 |-> <Nigeria>} ≡ [ <England>, <Germany>,
                                            <Colombia>, <Nigeria>]
```

4.2.3 Map Types

A map type from a type `A` to a type `B` is a type that associates with each element of `A` (or a subset of `A`) an element of `B`. A map value can be thought of as an unordered collection of pairs. The first element in each pair is called a key, because it can be used as a key to get the second element (called the information part) in that pair. All key elements in a map must therefore be unique. The set of all key elements is called the domain of the map, while the set of all information values is called the range of the map. All maps in VDM-SL are finite. The domain and range elements of a map type can be arbitrarily complex, they could e.g. be maps themselves.

A special kind of map is the injective map. An injective map is one for which no element of the range is associated with more than one element of the domain. For an injective map it is possible to invert the map.

In the following this convention will be used: `m`, `m1` and `m2` are maps from an arbitrary type `A` to another arbitrary type `B`, `ms` is a set of map values, `a`, `a1`, `a2` and `an` are elements from `A` while `b`, `b1`, `b2` and `bn` are elements from `B` and `P` is a logic predicate. `e1` and `e2` are arbitrary expressions and `s` is an arbitrary set.

Syntax: type = map type
| . . . ;

map type = general map type
| injective map type ;

general map type = ‘`map`’, type, ‘`to`’, type ;

injective map type = ‘`inmap`’, type, ‘`to`’, type ;

Equation: `M = map A to B` or `M = inmap A to B`

Constructors:

Map enumeration: `{a1 |-> b1, a2 |-> b2, ..., an |-> bn}` constructs a mapping of the enumerated maplets. The empty map will be written as `{|->}`.

Map comprehension: `{ed |-> er | bd1, ..., bdn & P}` constructs a mapping by evaluating the expressions `ed` and `er` on all the possible bindings for which the predicate `P` evaluates to `true`. `bd1, ..., bdn` are bindings of free identifiers from the expressions `ed` and `er` to sets or types.

The syntax and semantics of all map expressions are given in section 7.9.

Operators:

Operator	Name	Type
`dom m`	Domain	(map A to B) $\to$ set of A
`rng m`	Range	(map A to B) $\to$ set of B
`m1 munion m2`	Merge	(map A to B) $*$ (map A to B) $\to$ map A to B
`m1 ++ m2`	Override	(map A to B) $*$ (map A to B) $\to$ map A to B
`merge ms`	Distributed merge	set of (map A to B) $\to$ map A to B
`s <: m`	Domain restrict to	(set of A) $*$ (map A to B) $\to$ map A to B
`s <-: m`	Domain restrict by	(set of A) $*$ (map A to B) $\to$ map A to B
`m :> s`	Range restrict to	(map A to B) $*$ (set of B) $\to$ map A to B
`m :-> s`	Range restrict by	(map A to B) $*$ (set of B) $\to$ map A to B
`m(d)`	Map apply	(map A to B) $*$ $A \to B$
`m1 comp m2`	Map composition	(map B to C) $*$ (map A to B) $\to$ map A to C
`m ** n`	Map iteration	(map A to A) $*$ nat $\to$ map A to A
`m1 = m2`	Equality	(map A to B) $*$ (map A to B) $\to$ bool
`m1 <> m2`	Inequality	(map A to B) $*$ (map A to B) $\to$ bool
`inverse m`	Map inverse	inmap A to B $\to$ inmap B to A

Semantics of Operators: Two maps `m1` and `m2` are compatible if any common element of `dom m1` and `dom m2` is mapped to the same value by both maps.

Operator Name	Semantics Description
Domain	yields the domain (the set of keys) of `m`.
Range	yields the range (the set of information values) of `m`.
Merge	yields a map combined by `m1` and `m2` such that the resulting map maps the elements of `dom m1` as does `m1`, and the elements of `dom m2` as does `m2`. The two maps must be compatible.
Override	overrides and merges `m1` with `m2`, i.e. it is like a merge except that `m1` and `m2` need not be compatible; any common elements are mapped as by `m2` (so `m2` overrides `m1`).
Distributed merge	yields the map that is constructed by merging all the maps in `ms`. The maps in `ms` must be compatible.
Domain restricted to	creates the map consisting of the elements in `m` whose key is in `s`. `s` need not be a subset of `dom m`.

Operator Name	Semantics Description
Domain restricted by	creates the map consisting of the elements in `m` whose key is not in `s`. `s` need not be a subset of `dom m`.
Range restricted to	creates the map consisting of the elements in `m` whose information value is in `s`. `s` need not be a subset of `rng m`.
Range restricted by	creates the map consisting of the elements in `m` whose information value is not in `s`. `s` need not be a subset of `rng m`.
Map apply	yields the information value whose key is `d`. `d` must be in the domain of `m`.
Map composition	yields the the map that is created by composing `m2` elements with `m1` elements. The resulting map is a map with the same domain as `m2`. The information value corresponding to a key is the one found by first applying `m2` to the key and then applying `m1` to the result. `rng m2` must be a subset of `dom m1`.
Map iteration	yields the map where `m` is composed with itself `n` times. `n=0` yields the identity map where each element of `dom m` is map into itself; `n=1` yields `m` itself. For `n>1`, the range of `m` must be a subset of `dom m`.
Map inverse	yields the inverse map of `m`. `m` must be a 1-to-1 mapping.

Examples: Let

```
m1 = { <France> |-> 9, <Denmark> |-> 4,
       <SouthAfrica> |-> 2, <SaudiArabia> |-> 1},
m2 = { 1 |-> 2, 2 |-> 3, 3 |-> 4, 4 |-> 1 },
Europe = { <France>, <England>, <Denmark>, <Spain> }
```

then:

```
dom m1          ≡  {<France>, <Denmark>,
                    <SouthAfrica>,
                    <SaudiArabia>}

rng m1          ≡  {1,2,4,9}
```

```
m1 munion {<England> |-> 3}            ≡  {<France> |-> 9,
                                           <Denmark> |-> 4,
                                           <England> |-> 3,
                                           <SaudiArabia> |-> 1,
                                           <SouthAfrica> |-> 2}

m1 ++ {<France> |-> 8,                 ≡  {<France> |-> 8,
       <England> |-> 4}                    <Denmark> |-> 4,
                                           <SouthAfrica> |-> 2,
                                           <SaudiArabia> |-> 1,
                                           <England> |-> 4}

merge{ {<France> |-> 9,                ≡  {<France> |-> 9,
        <Spain> |-> 4}                     <England> |-> 3,
       {<France> |-> 9,                    <Spain> |-> 4,
        <England> |-> 3,                   <UnitedStates> |-> 1}
        <UnitedStates> |-> 1}}

Europe <: m1                           ≡  {<France> |-> 9,
                                           <Denmark> |-> 4}

Europe <-: m1                          ≡  {<SouthAfrica> |-> 2,
                                           <SaudiArabia> |-> 1}

m1 :> {2,...,10}                       ≡  {<France> |-> 9,
                                           <Denmark> |-> 4,
                                           <SouthAfrica> |-> 2}

m1 :-> {2,...,10}                      ≡  {<SaudiArabia> |-> 1}

m1 comp ({"France" |-> <France>})      ≡  {"France" |-> 9}

m2 ** 3                                ≡  {1 |-> 4, 2 |-> 1,
                                           3 |-> 2, 4 |-> 3 }

inverse m2                             ≡  {2 |-> 1, 3 |-> 2,
                                           4 |-> 3, 1 |-> 4 }

m2 comp (inverse m2)                   ≡  {1 |-> 1, 2 |-> 2,
                                           3 |-> 3, 4 |-> 4 }
```

4.2.4 Product Types

The values of a product type are called tuples. A tuple is a fixed length list where the i'th element of the tuple must belong to the i'th element of the product type.

Syntax: type = product type
| ... ;

product type = type, '*', type, { '*', type } ;

A product type consists of at least two subtypes.

Equation: `T = A1 * A2 * ... * An`

Constructors: The tuple constructor: `mk_(a1, a2, ..., an)`

The syntax and semantics for the tuple constructor are given in section 7.10.

Operators:

Operator	Name	Type
`t.#n`	Select	$T * \mathsf{nat} \rightarrow Ti$
`t1 = t2`	Equality	$T * T \rightarrow \mathsf{bool}$
`t1 <> t2`	Inequality	$T * T \rightarrow \mathsf{bool}$

The only operators working on tuples are component select, equality and inequality. Tuple components may be accessed using the select operator or by matching against a tuple pattern. Details of the semantics of the tuple select operator and an example of its use are given in section 7.12.

Examples: Let `a = mk_(1, 4, 8)`, `b = mk_(2, 4, 8)` then:

`a = b`	$\equiv$	false
`a <> b`	$\equiv$	true
`a = mk_(2,4)`	$\equiv$	false

4.2.5 Composite Types

Composite types correspond to record types in programming languages. Thus, elements of this type are somewhat similar to the tuples described in the section about product types above. The difference between the record type and the product type is that the different components of a record can be directly selected by means of corresponding selector functions. In addition records are tagged with an identifier which must be used when manipulating the record. The only way to tag a type is by defining it as a record. It is therefore common usage to define records with only one field in order to give it a tag. This is another difference to tuples as a tuple must have at least two entries whereas records can be empty.

In VDM-SL, is_ is a reserved prefix for names and it is used in an *is expression*. This is a built-in operator which is used to determine which record type a record value belongs to. It is often used to discriminate between the subtypes of a union type and will therefore be explained further in section 4.2.6. In addition to record types the is_ operator can also determine if a value is of one of the basic types.

In the following this convention will be used: `A` is a record type, `A1, ..., Am` are arbitrary types, `r`, `r1`, and `r2` are record values, `i1, ..., im` are selectors from the `r` record value, `e1, ..., em` are arbitrary expressions.

Syntax: type = composite type
| ... ;

composite type = 'compose', identifier, 'of', field list, 'end' ;

field list = { field } ;

field = [identifier, ':'], type
| [identifier, ':-'], type ;

or the shorthand notation

composite type = identifier, '::', field list ;

where identifier denotes both the type name and the tag name.

Equation:

```
A :: selfirst : A1
     selsec   : A2
```

or

```
A :: selfirst :  A1
     selsec   :- A2
```

or

```
A :: A1 A2
```

In the second notation, an *equality abstraction* field is used for the second field `selsec`. The minus indicates that such a field is ignored when comparing records using the equality operator. In the last notation the fields of `A` can only be accessed by pattern matching (like it is done for tuples) as the fields have not been named.

In the last notation the fields of `A` can only be accessed by pattern matching (as is done for tuples) since the fields have not been named.

The shorthand notation `::` used in the two previous examples where the tag name equals the type name, is the notation most used. The more general compose notation is typically used if a composite type has to be specified directly as a component of a more complex type:

```
T = map S to compose A of A1 A2 end
```

It should be noted however that composite types can only be used in type definitions, and not e.g. in signatures to functions or operations.

Typically composite types are used as alternatives in a union type definition (see 4.2.6) such as:

```
MasterA = A | B | ...
```

where `A` and `B` are defined as composite types themselves. In this situation the is_ predicate can be used to distingush the alternatives.

Constructors: The record constructor: `mk_A(a, b)` where `a` belongs to the type `A1` and `b` belongs to the type `A2`.

The syntax and semantics for all record expressions are given in section 7.11.

Operators:

Operator	Name	Type
`r.i`	Field select	$A * Id \rightarrow Ai$
`r1 = r2`	Equality	$A * A \rightarrow$ bool
`r1 <> r2`	Inequality	$A * A \rightarrow$ bool
`is_A(r1)`	Is	$Id * MasterA \rightarrow$ bool

Semantics of Operators:

Operator Name	Semantics Description
Field select	yields the value of the field with fieldname `i` in the record value `r`. `r` must have a field with name `i`.

Examples: Let `Score` be defined as

```
Score :: team : Team
         won : nat
         drawn : nat
         lost : nat
         points : nat;
Team = <Brazil> | <France> | ...
```

and let

```
sc1 = mk_Score (<France>, 3, 0, 0, 9),
sc2 = mk_Score (<Denmark>, 1, 1, 1, 4),
sc3 = mk_Score (<SouthAfrica>, 0, 2, 1, 2) and
sc4 = mk_Score (<SaudiArabia>, 0, 1, 2, 1).
```

Then

```
sc1.team                  ≡  <France>
sc4.points                ≡  1
sc2.points > sc3.points   ≡  true
is_Score(sc4)             ≡  true
is_bool(sc3)              ≡  false
is_int(sc1.won)           ≡  true
sc4 = sc1                 ≡  false
sc4 <> sc2                ≡  true
```

The equality abstraction field, written using ‘`:-`’ instead of ‘`:`’, may be useful, for example, when working with lower level models of an abstract syntax of a programming language. For example, one may wish to add a position information field to a type of identifiers without affecting the true identity of identifiers:

```
Id :: name :  seq of char
      pos  :- nat
```

The effect of this will be that the `pos` field is ignored in equality comparisons, e.g. the following would evaluate to true:

```
mk_Id("x",7) = mk_Id("x",9)
```

In particular this can be useful when looking up in an environment which is typically modelled as a map of the following form:

```
Env = map Id to Val
```

Such a map will contain at most one index for a specific identifier, and a map lookup will be independent of the `pos` field.

Moreover, the equality abstraction field will affect set expressions. For example,

```
{mk_Id("x",7),mk_Id("y",8),mk_Id("x",9)}
```

will be equal to

```
{mk_Id("x",?),mk_Id("y",8)}
```

where the question mark stands for 7 or 9.

Finally, note that for equality abstraction fields valid patterns are limited to don't care and identifier patterns. Since equality abstraction fields are ignored when comparing two values, it does not make sense to use more complicated patterns.

4.2.6 Union and Optional Types

The union type corresponds to a set-theoretic union, i.e. the type defined by means of a union type will contain all the elements from each of the components of the union type. It is possible to use types that are not disjoint in the union type, even though such usage would be bad practice. However, the union type is normally used when something belongs to one type from a set of possible types. The types which constitute the union type are often composite types. This makes it possible, using the is_ operator, to decide which of these types a given value of the union type belongs to.

The optional type `[T]` is a kind of shorthand for a union type `T | nil`, where nil is used to denote the absence of a value. However, it is not possible to use the set {nil} as a type so the only types nil will belong to will be optional types.

Syntax: type = union type
| optional type
| ... ;

union type = type, ‘|’, type, { ‘|’, type } ;

optional type = ‘[’, type, ‘]’ ;

Equation: `B = A1 | A2 | ... | An`

Constructors: None.

Operators:

Operator	Name	Type
`t1 = t2`	Equality	$A * A \to$ bool
`t1 <> t2`	Inequality	$A * A \to$ bool

Examples: In this example `Expr` is a union type whereas `Const`, `Var`, `Infix` and `Cond` are composite types defined using the shorthand `::` notation.

```
Expr  = Const | Var | Infix | Cond;
Const :: nat | bool;
Var   :: id:Id
         tp: [<Bool> | <Nat>];
Infix :: Expr * Op * Expr;
Cond  :: test : Expr
         cons : Expr
         altn : Expr
```

and let `expr = mk_Cond(mk_Var("b",<Bool>),mk_Const(3), mk_Var("v",nil))` then:

```
is_Cond(expr)         ≡  true
is_Const(expr.cons)   ≡  true
is_Var(expr.altn)     ≡  true
is_Infix(expr.test)   ≡  false
```

Using union types we can extend the use of previously defined operators. For instance, interpreting `=` as a test over `bool | nat` we have

```
1 = false  ≡  false
```

Similarly we can take use union types for taking unions of sets and concatenating sequences:

```
{1,2} union {false,true}  ≡  {1,2, false,true}
['a','b']^[<c>,<d>]       ≡  ['a','b', <c>,<d>]
```

In the set union, we take the union over sets of type `nat | bool`; for the sequence concatenation we are manipulating sequences of type `char | <c> | <d>`.

4.2.7 Function Types

In VDM-SL function types can also be used in type definitions. A function type from a type `A` (actually a list of types) to a type `B` is a type that associates with each element of `A` an element of `B`. A function value can be thought of as a function in a programming language which has no side-effects (i.e. it does not use any global variables).

Such usage can be considered advanced in the sense that functions are used as values (thus this section may be skipped during the first reading). Function values may be created by lambda expressions (see below), or by function definitions, which are described in section 6. Function values can be of higher order in the sense that they can take functions as arguments or return functions as results. In this way functions can be Curried such that a new function is returned when the first set of parameters are supplied (see the examples below).

Syntax: type = partial function type
| … ;

function type = partial function type
| total function type ;

partial function type = discretionary type, '->', type ;

total function type = discretionary type, '+>', type ;

discretionary type = type | '(',')' ;

Equation: `F = A +> B`[11] or `F = A -> B`

Constructors: In addition to the traditional function definitions the only way to construct functions is by the lambda expression: `lambda pat1 : T1, ..., patn : Tn & body` where the `patj` are patterns, the `Tj` are type expressions, and `body` is the body expression which may use the pattern identifiers from all the patterns.

The syntax and semantics for the lambda expression are given in section 7.13.

[11] Note that the total function arrow can only be used in signatures of totally defined functions and thus not in a type definition.

Operators:

Operator	Name	Type
`f(a1,...,an)`	Function apply	$A1 * \cdots * An \rightarrow B$
`f1 comp f2`	Function composition	$(B \rightarrow C) * (A \rightarrow B) \rightarrow (A \rightarrow C)$
`f ** n`	Function iteration	$(A \rightarrow A) * \mathsf{nat} \rightarrow (A \rightarrow A)$
`t1 = t2`	Equality	$A * A \rightarrow \mathsf{bool}$
`t1 <> t2`	Inequality	$A * A \rightarrow \mathsf{bool}$

Note that equality and inequality between type values should be used with great care. In VDM-SL this corresponds to the mathematical equality (and inequality) which is not computable for infinite values like general functions. Thus, in the interpreter the equality is on the abstract syntax of the function value (see `inc1` and `inc2` below).

Semantics of Operators:

Operator Name	Semantics Description
Function apply	yields the result of applying the function `f` to the values of $\mathtt{a}_j$. See the definition of apply expressions in Section 7.12.
Function composition	it yields the function equivalent to applying first `f2` and then applying `f1` to the result. `f1`, but not `f2` may be Curried.
Function iteration	yields the funciton equivalent to applying `f` `n` times. `n=0` yields the identity function which just returns the value of its parameter; `n=1` yields the function itself. For `n>1`, the result of `f` must be contained in its parameter type.

Examples: Let the following function values be defined:

```
f1 = lambda x : nat & lambda y : nat & x + y
f2 = lambda x : nat & x + 2
inc1 = lambda x : nat & x + 1
inc2 = lambda y : nat & y + 1
```

then the following holds:

```
f1(5)          ≡  lambda y :nat & 5 + y
f2(4)          ≡  6
f1 comp f2     ≡  lambda x :nat & lambda y :nat & (x + 2) + y
f2 ** 4        ≡  lambda x :nat & x + 8
inc1 = inc2    ≡  false
```

Notice that the equality test does not yield the expected result with respect to the semantics of VDM-SL . Thus, one should be **very** careful with the usage of equality for infinite values like functions.

4.3 Invariants

If the data types specified by means of equations as described above contain values which should not be allowed, then it is possible to restrict the values in a type by means of an invariant. The result is that the type is restricted to a subset of its original values. Thus, by means of a predicate the acceptable values of the defined type are limited to those where this expression is true.

The general scheme for using invariants looks like this:

```
Id = Type
inv pat == expr
```

where `pat` is a pattern matching the values belonging to the type `Id`, and `expr` is a truth-valued expression, involving some or all of the identifiers from the pattern `pat`.

If an invariant is defined, a new (total) function is implicitly created with the signature:

```
inv_Id : Type +> bool
```

This function can be used within other invariant, function or operation definitions.

For instance, recall the record type `Score` defined on page 25. We can ensure that the number of points awarded is consistent with the number of games won and drawn using an invariant:

```
Score :: team : Team
         won : nat
```

```
          drawn : nat
          lost : nat
          points : nat
  inv sc == sc.points = 3 * sc.won + sc.drawn;
```

The invariant function implicitly created for this type is:

```
  inv_Score : Score +> bool
  inv_Score (sc) ==
    sc.points = 3 * sc.won + sc.drawn;
```

5 Algorithm Definitions

In VDM-SL algorithms can be defined by both functions and operations. However, they do not directly correspond to functions in traditional programming languages. What separates functions from operations in VDM-SL is the use of local and global variables. Operations can manipulate both the global variables and any local variables. Both local and global variables will be described later. Functions are pure in the sense that they cannot access global variables and they are not allowed to define local variables. Thus, functions are purely applicative while operations are imperative.

Functions and operations can be defined both explicitly (by means of an explicit algorithm definition) or implicitly (by means of a pre-condition and/or a post condition). An explicit algorithm definition for a function is called an expression while for an operation it is called a statement. A pre-condition is a truth-valued expression which specifies what must hold before the function/operation is evaluated. A pre-condition can only refer to parameter values and global variables (if it is an operation). A post-condition is also a truth valued expression which specifies what must hold after the function/operation is evaluated. A post-condition can refer to the result identifier, the parameter values, the current values of global variables and the old values of global variables. The old values of global variables are the values of the variables as they were before the operation was evaluated. Only operations can refer to the old values of global variables in a post-condition as functions are not allowed to change the global variables.

However, in order to be able to execute both functions and operations by the interpreter they must be defined explicitly[12]. In VDM-SL it is also possible for

[12]Implicitly specified functions and operations cannot in general be executed because their

explicit function and operation definitions to specify an additional pre- and a post-condition. In the post-condition of explicit function and operation definitions the result value must be referred to by the reserved word RESULT.

6 Function Definitions

In VDM-SL we can define first order and higher order functions. A higher order function is either a Curried function (a function that returns a function as result), or a function that takes functions as arguments. Furthermore, both first order and higher order functions can be polymorphic. In general, the syntax for the definition of a function is:

function definitions = 'functions', [function definition,
{ ';', function definition }, [';']] ;

function definition = explicit function definition
| implicit function definition
| extended explicit function definition ;

explicit function definition = identifier,
[type variable list], ':', function type,
identifier, parameters list, '==',
function body,
['pre', expression],
['post', expression],
['measure', name] ;

implicit function definition = identifier, [type variable list],
parameter types, identifier type pair list,
['pre', expression],
'post', expression ;

extended explicit function definition = identifier, [type variable list],
parameter types,

post-condition does not need to directly relate the output to the input. Often it is done by specifying the properties the output must satisfy.

identifier type pair list,
'==', function body,
['pre', expression],
['post', expression] ;

type variable list = '[', type variable identifier,
{ ',', type variable identifier }, ']' ;

identifier type pair list = identifier, ':', type,
{ ',', identifier, ':', type } ;

parameter types = '(', [pattern type pair list], ')' ;

pattern type pair list = pattern list, ':', type,
{ ',', pattern list,':', type } ;

function type = partial function type
| total function type ;

partial function type = discretionary type, '->', type ;

total function type = discretionary type, '+>', type ;

discretionary type = type | '(',')' ;

parameters = '(', [pattern list], ')' ;

pattern list = pattern,{ ',', pattern } ;

function body = expression
| 'is not yet specified' ;

Here is not yet specified may be used as the function body during development of a model.

A simple example of an explicit function definition is the function `map_inter` which takes two compatible maps over natural numbers and returns those maplets common to both

```
map_inter: (map nat to nat) * (map nat to nat) -> map nat to nat
map_inter (m1,m2) ==
  (dom m1 inter dom m2) <: m1
pre  forall d in set dom m1 inter dom m2 & m1(d) = m2(d)
```

Note that we could also use the optional post condition to allow assertions about the result of the function:

```
map_inter: (map nat to nat) * (map nat to nat) -> map nat to nat
map_inter (m1,m2) ==
  (dom m1 inter dom m2) <: m1
pre  forall d in set dom m1 inter dom m2 & m1(d) = m2(d)
post  dom RESULT = dom m1 inter dom m2
```

The same function can also be defined implicitly:

```
map_inter2 (m1,m2: map nat to nat) m: map nat to nat
pre  forall d in set dom m1 inter dom m2 & m1(d) = m2(d)
post  dom m = dom m1 inter dom m2 and
      forall d in set dom m & m(d) = m1(d);
```

A simple example of an extended explicit function definition (non-standard) is the function `map_disj` which takes a pair of compatible maps over natural numbers and returns the map consisting of those maplets unique to one or other of the given maps:

```
map_disj (m1:map nat to nat,m2:map nat to nat) res : map nat to nat ==
  (dom m1 inter dom m2) <-: m1 munion
  (dom m1 inter dom m2) <-: m2
pre  forall d in set dom m1 inter dom m2 & m1(d) = m2(d)
post  dom res = (dom m1 union dom m2) \ (dom m1 inter dom m2)
      and
      forall d in set dom res & res(d) = m1(d) or res(d) = m2(d)
```

(Note here that an attempt to interpret the post-condition could potentially result in a run-time error since `m1(d)` and `m2(d)` need not both be defined simultaneously.)

The functions `map_inter` and `map_disj` can be evaluated by the interpreter, but the implicit function `map_inter2` cannot be evaluated. However, in all three cases the pre- and post-conditions can be used in other functions; for instance from the definition of `map_inter2` we get functions `pre_map_inter2` and `post_map_inter2` with the following signatures:

```
pre_map_inter2 : (map nat to nat) * (map nat to nat) +> bool
post_map_inter2 : (map nat to nat) * (map nat to nat) *
                  (map nat to nat) +> bool
```

These kinds of functions are automatically created by the interpreter and they can be used in other definitions (this technique is called quoting). In general, for a function `f` with signature

```
f : T1 * ... * Tn -> Tr
```

defining a pre-condition for the function causes creation of a function `pre_f` with signature

```
pre_f : T1 * ... * Tn +> bool
```

and defining a post-condition for the function causes creation of a function `post_f` with signature

```
post_f : T1 * ... * Tn * Tr +> bool
```

Functions can also be defined using recursion (i.e. by calling themselves). When this is done one is recommended to add a 'measure' function that can be used in the proof obligations generated from the model such that termination proofs can be carried out. A simple example here could be the traditional factorial function defined as:

```
functions

fac: nat +> nat
fac(n) ==
  if n = 0
  then 1
  else n * fac(n - 1)
measure id
```

where `id` would be defined as:

```
id: nat +> nat
id(n) == n
```

6.1 Polymorphic Functions

Functions can also be polymorphic. This means that we can create generic functions that can be used on values of several different types. For this purpose type parameters (or type variables which are written like normal identifiers prefixed with a `@` sign) are used. Consider the polymorphic function to create an empty bag:[13]

```
empty_bag[@elem] : () +> (map @elem to nat1)
empty_bag() ==
  { |-> }
```

Before we can use the above function, we have to instantiate the function `empty_bag` with a type, for example integers (see also section 7.12):

```
emptyInt = empty_bag[int]
```

Now we can use the function `emptyInt` to create a new bag to store integers. More examples of polymorphic functions are:

```
num_bag[@elem] : @elem * (map @elem to nat1) +> nat
num_bag(e, m) ==
  if e in set dom m
  then m(e)
  else 0;

plus_bag[@elem] : @elem * (map @elem to nat1) +> (map @elem to nat1)
plus_bag(e, m) ==
  m ++ { e |-> num_bag[@elem](e, m) + 1 }
```

[13]The examples for polymorphic functions are taken from [2]. Bags are modelled as maps from the elements to their multiplicity in the bag. The multiplicity is at least 1, i.e. a non-element is not part of the map, rather than being mapped to 0.

If pre- and or post-conditions are defined for polymorphic functions, the corresponding predicate functions are also polymorphic. For instance if num_bag was defined as

```
num_bag[@elem] : @elem * (map @elem to nat1) +> nat
num_bag(e, m) ==
  m(e)
pre  e  in set dom  m
```

then the pre-condition function would be

```
pre_num_bag[@elem] :@elem * (map @elem to nat1) +> bool
```

In case functions are defined polymorphic a measure should also be used.

6.2 Higher Order Functions

Functions are allowed to receive other functions as arguments. A simple example of this is the function nat_filter which takes a sequence of natural numbers, and a predicate, and returns the subsequence that satisfies this predicate:

```
nat_filter : (nat -> bool) * seq of nat -> seq of nat
nat_filter (p,ns) ==
  [ns(i) | i in set inds ns & p(ns(i))];
```

Then nat_filter (lambda x:nat & x mod 2 = 0, [1,2,3,4,5]) ≡ [2,4]. In fact, this algorithm is not specific to natural numbers, so we may define a polymorphic version of this function:

```
filter[@elem]: (@elem -> bool) * seq of @elem -> seq of @elem
filter (p,l) ==
  [l(i) | i in set inds l & p(l(i))];
```

so filter[real](lambda x:real & floor x = x, [2.3,0.7,-2.1,3]) ≡ [3].

Functions may also return functions as results. An example of this is the function fmap:

```
fmap[@elem]: (@elem -> @elem) -> seq of @elem -> seq of @elem
fmap (f)(l) ==
  if l = []
  then []
  else [f(hd l)]^(fmap[@elem] (f)(tl l));
```

So `fmap[nat](lambda x:nat & x * x)([1,2,3,4,5])` ≡ `[ 1,4,9,16,25 ]`

7 Expressions

In this subsection we will describe the different kinds of expressions one by one. Each of them will be described by means of:

- A syntax description in BNF.
- An informal semantics description.
- An example illustrating its usage.

7.1 Let Expressions

Syntax:

```
expression = let expression
           | let be expression
           | ... ;

let expression = 'let', local definition { ',', local definition },
                 'in', expression ;

let be expression = 'let', bind, [ 'be', 'st', expression ], 'in',
                    expression ;

local definition = value definition
                 | function definition ;

value definition = pattern, [ ':', type ], '=', expression ;
```

where the "function definition" component is described in section 6.

Semantics: A simple *let expression* has the form:

```
let p1 = e1, ..., pn = en in e
```

where `p1, ..., pn` are patterns, `e1, ..., en` are expressions which match the corresponding pattern `pi`, and `e` is an expression, of any type, involving the pattern identifiers of `p1, ..., pn`. It denotes the value of the expression `e` in the context in which the patterns `p1, ..., pn` are matched against the corresponding expressions `e1, ..., en`.

More advanced let expressions can also be made by using local function definitions. The semantics of doing so is simply that the scope of such locally defined functions is restricted to the body of the let expression.

In standard VDM-SL the collection of definitions may be mutually recursive. However, in VDM-SL this is not supported by the interpreter. Furthermore, the definitions must be ordered such that all constructs are defined before they are used.

A *let-be-such-that expression* has the form:

```
let b be st e1 in e2
```

where `b` is a binding of a pattern to a set value (or a type), `e1` is a boolean expression, and `e2` is an expression, of any type, involving the pattern identifiers of the pattern in `b`. The `be st e1` part is optional. The expression denotes the value of the expression `e2` in the context in which the pattern from `b` has been matched against either an element in the set from `b` or against a value from the type in `b`[14]. If the `st e1` expression is present, only such bindings where `e1` evaluates to true in the matching context are used.

Examples: *Let expressions* are useful for improving readability especially by contracting complicated expressions used more than once. For instance, we can improve the function `map_disj` from page 34:

```
map_disj : (map nat to nat) * (map nat to nat) -> map nat to nat
map_disj (m1,m2) ==
  let inter_dom = dom m1 inter dom m2
  in
    inter_dom <-: m1 munion
    inter_dom <-: m2
pre forall d in set dom m1 inter dom m2 & m1(d) = m2(d)
```

[14]Remember that only the set bindings can be executed by means of the interpreter.

They are also convenient for decomposing complex structures into their components. For instance, using the previously defined record type `Score` (page 25) we can test whether one score is greater than another:

```
let mk_Score(-,w1,-,-,p1) = sc1,
    mk_Score(-,w2,-,-,p2) = sc2
in (p1 > p2) or (p1 = p2 and w1 > w2)
```

In this particular example we extract the second and fifth components of the two scores. Note that don't care patterns (page 64) are used to indicate that the remaining components are irrelevant for the processing done in the body of this expression.

Let-be-such-that expressions are useful for abstracting away the non-essential choice of an element from a set, in particular in formulating recursive definitions over sets. An example of this is a version of the sequence filter function (page 37) over sets:

```
set_filter[@elem] : (@elem -> bool) -> (set of @elem) ->
                    (set of @elem)
set_filter(p)(s) ==
  if s = {}
  then {}
  else let x in set s
       in (if p(x) then {x} else {}) union
          set_filter[@elem](p)(s \ {x});
```

We could alternatively have defined this function using a set comprehension (described in section 7.7):

```
set_filter[@elem] : (@elem -> bool) -> (set of @elem) ->
                    (set of @elem)
set_filter(p)(s) ==
  { x | x in set s & p(x)};
```

The last example shows how the optional "be such that" part (be st) can be used. This part is especially useful when it is known that an element with some property exists but an explicit expression for such an element is not known or difficult to write. For instance we can exploit this expression to write a selection sort algorithm:

```
remove : nat * seq of nat -> seq of nat
remove (x,l) ==
  let i in set inds l be st l(i) = x
  in l(1,...,i-1)^l(i+1,...,len l)
pre  x in set elems l;

selection_sort : seq of nat -> seq of nat
selection_sort (l) ==
  if l = []
  then []
  else let m in set elems l be st
       forall x in set elems l & m <= x
       in [m]^(selection_sort (remove(m,l)))
```

Here the first function removes a given element from the given list; the second function repeatedly removes the least element in the unsorted portion of the list, and places it at the head of the sorted portion of the list.

7.2 The Define Expression

This expression can only be used inside operations which will be described in section 12. In order to deal with global variables inside the expression part an extra expression construct is available inside operations.

Syntax: expression = ...
| def expression
| ... ;

def expression = 'def', pattern bind, '=', expression,
{ ';', pattern bind, '=', expression }, [';'],
'in', expression ;

Semantics: A *define expression* has the form:

```
def pb1 = e1;
    ...
    pbn = en
in
  e
```

The *define expression* corresponds to a let expression except that the right hand side expressions may depend on the value of the local and/or global variable and that it may not be mutually recursive. It denotes the value of the expression `e` in the context in which the patterns (or binds) `pb1, ..., pbn` are matched against the corresponding expressions `e1, ..., en`[15].

Examples: The *define expression* is used in a pragmatic way, in order to make the reader aware of the fact that the value of the expression depends upon the global variable.

This can be illustrated by a small example:

```
def  user = lib(copy) in
  if user = <OUT>
  then  true
  else  false
```

where `copy` is defined in the context, `lib` is global variable (thus `lib(copy)` can be considered as looking up the contents of a part of the variable).

The operation `GroupRunnerUp_expl` in section 13.1 also gives an example of a define expression.

7.3 Unary and Binary Expressions

Syntax: expression = ...
| unary expression
| binary expression
| ... ;

unary expression = prefix expression
| map inverse ;

prefix expression = unary operator, expression ;

unary operator = '+' | '−' | 'abs' | 'floor' | 'not'
| 'card' | 'power' | 'dunion' | 'dinter'
| 'hd' | 'tl' | 'len' | 'elems' | 'inds' | 'conc'
| 'dom' | 'rng' | 'merge' ;

[15] If binds are used, it simply means that the values which can match the pattern are further constrained by the type or set expression as explained in section 8.

map inverse = ‘inverse’, expression ;

binary expression = expression, binary operator, expression ;

binary operator = ‘+’ | ‘-’ | ‘*’ | ‘/’
| ‘rem’ | ‘div’ | ‘mod’ | ‘**’
| ‘union’ | ‘inter’ | ‘\’ | ‘subset’
| ‘psubset’ | ‘in set’ | ‘not in set’
| ‘^’
| ‘++’ | ‘munion’ | ‘<:’ | ‘<-:’ | ‘:>’ | ‘:->’
| ‘and’ | ‘or’
| ‘=>’ | ‘<=>’ | ‘=’ | ‘<>’
| ‘<’ | ‘<=’ | ‘>’ | ‘>=’
| ‘comp’ ;

Semantics: Unary and binary expressions are a combination of operands and operators denoting a value of a specific type. The signature of all these operators is already given in section 4, so no further explanation will be provided here. The map inverse unary operator is treated separately because it is written with postfix notation in the mathematical syntax.

Examples: Examples using these operators were given in section 4, so none will be provided here.

7.4 Conditional Expressions

Syntax: expression = ...
| if expression
| cases expression
| ... ;

if expression = ‘if’, expression, ‘then’, expression,
{ elseif expression }, ‘else’, expression ;

elseif expression = ‘elseif’, expression, ‘then’, expression ;

cases expression = ‘cases’, expression, ‘:’,
cases expression alternatives,
[‘,’, others expression], ‘end’ ;

cases expression alternatives = cases expression alternative,
{ ‘,’, cases expression alternative } ;

cases expression alternative = pattern list, ‘->’, expression ;

others expression = ‘others’, ‘->’, expression ;

Semantics: *If expressions* and *cases expressions* allow the choice of one from a number of expressions on the basis of the value of a particular expression.

The *if expression* has the form:

```
if e1
then  e2
else  e3
```

where e1 is a boolean expression, while e2 and e3 are expressions of any type. The if expression denotes the value of e2 evaluated in the given context if e1 evaluates to true in the given context. Otherwise the if expression denotes the value of e3 evaluated in the given context. The use of an elseif expression is simply a shorthand for a nested if then else expression in the else part of the expression.

The *cases expression* has the form

```
cases e :
  p11, p12, ..., p1n -> e1,
  ...                -> ...,
  pm1, pm2, ..., pmk -> em,
  others             -> emplus1
end
```

where e is an expression of any type, all pij’s are patterns which are matched one by one against the expression e. The ei’s are expressions of any type, and the keyword others and the corresponding expression emplus1 are optional. The cases expression denotes the value of the ei expression evaluated in the context in which one of the pij patterns has been matched against e. The chosen ei is the first entry where it has been possible to match the expression e against one of the patterns. If none of the patterns match e an others clause must be present, and then the cases expression denotes the value of emplus1 evaluated in the given context.

Examples: The if expression in VDM-SL corresponds to what is used in most programming languages, while the cases expression in VDM-SL is more general than most programming languages. This is shown by the fact that real pattern matching is taking place, but also because the patterns do not have to be constants as in most programming languages.

An example of the use of conditional expressions is provided by the specification of the mergesort algorithm:

```
lmerge : seq of nat * seq of nat -> seq of nat
lmerge (s1,s2) ==
  if s1 = [] then s2
  elseif s2 = [] then s1
  elseif (hd s1) < (hd s2)
  then [hd s1]^(lmerge (tl s1, s2))
  else [hd s2]^(lmerge (s1, tl s2));

mergesort : seq of nat -> seq of nat
mergesort (l) ==
  cases l:
    [] -> [],
    [x] -> [x],
    l1^l2 -> lmerge (mergesort(l1), mergesort(l2))
  end
```

The pattern matching provided by cases expressions is useful for manipulating members of type unions. For instance, using the type definition `Expr` from page 27 we have:

```
print_Expr : Expr -> seq1 of char
print_Expr (e) ==
  cases e:
    mk_Const(-) -> "Const of"^(print_Const(e)),
    mk_Var(id,-) -> "Var of"^id,
    mk_Infix(mk_(e1,op,e2)) -> "Infix of"^print_Expr(e1)^","
                                ^print_Op(op)^","
                                ^print_Expr(e2),
    mk_Cond(t,c,a) -> "Cond of"^print_Expr(t)^","
                           ^print_Expr(c)^","
                           ^print_Expr(a)
  end;

print_Const : Const -> seq1 of char
```

```
print_Const(mk_Const(c)) ==
  if is_nat(c)
  then "nat"
  else -- must be bool
       "bool";
```

The function `print_Op` would be defined similarly.

7.5 Quantified Expressions

Syntax: expression = ...
| quantified expression
| ... ;

quantified expression = all expression
| exists expression
| exists unique expression ;

all expression = 'forall', bind list, '&', expression ;

exists expression = 'exists', bind list, '&', expression ;

bind list = multiple bind, { ',', multiple bind } ;

exists unique expression = 'exists1', bind, '&', expression ;

Semantics: There are three forms of quantified expressions: *universal* (written as forall), *existential* (written as exists), and *unique existential* (written as exists1). Each yields a boolean value true or false, as explained in the following.

The *universal quantification* has the form:

```
forall mbd1, mbd2, ..., mbdn & e
```

where each `mbdi` is a multiple bind `pi in set s` (or if it is a type bind `pi : type`), and `e` is a boolean expression involving the pattern identifiers of the `mbdi`'s. It has the value true if `e` is true when evaluated in the context of every choice of bindings from `mbd1, mbd2, ..., mbdn` and false otherwise.

The *existential quantification* has the form:

```
exists mbd1, mbd2, ..., mbdn & e
```

where the `mbdi`'s and the `e` are as for a universal quantification. It has the value `true` if `e` is `true` when evaluated in the context of at least one choice of bindings from `mbd1, mbd2, ..., mbdn`, and `false` otherwise.

The *unique existential quantification* has the form:

```
exists1 bd & e
```

where `bd` is either a set bind or a type bind and `e` is a boolean expression involving the pattern identifiers of `bd`. It has the value `true` if `e` is `true` when evaluated in the context of exactly one choice of bindings, and `false` otherwise.

All quantified expressions have the lowest possible precedence. This means that the longest possible constituent expression is taken. The expression is continued to the right as far as it is syntactically possible.

Examples: An example of an existential quantification is given in the function shown below, `QualificationOk`. This function, taken from the specification of a nuclear tracking system in [3], checks whether a set of experts has a required qualification.

```
types

ExpertId = token;
Expert :: expertid : ExpertId
          quali : set of Qualification
inv ex == ex.quali <> ;
Qualification = <Elec> | <Mech> | <Bio> | <Chem>

functions

QualificationOK: set of Expert * Qualification -> bool
QualificationOK(exs,reqquali) ==
    exists ex in set exs & reqquali in set ex.quali
```

The function `min` gives us an example of a universal quantification:

```
min(s:set of nat) x:nat
pre  s <> {}
post  x in set s and
      forall y in set s \ {x} & y < x
```

We can use unique existential quantification to state the functional property satisfied by all maps `m`:

```
forall d in set dom m &
  exists1 r in set rng m & m(d) = r
```

7.6 The Iota Expression

Syntax: expression = ...
| iota expression
| ... ;

iota expression = 'iota', bind, '&', expression ;

Semantics: An *iota expression* has the form:

```
iota bd & e
```

where `bd` is either a set bind or a type bind, and `e` is a boolean expression involving the pattern identifiers of `bd`. The `iota` operator can only be used if a unique value exists which matches the bind and makes the body expression `e` yield `true` (i.e. `exists1 bd & e` must be `true`). The semantics of the iota expression is such that it returns the unique value which satisfies the body expression (`e`).

Examples: Using the values `sc1,...,sc4` defined by

```
sc1 = mk_Score (<France>, 3, 0, 0, 9);
sc2 = mk_Score (<Denmark>, 1, 1, 1, 4);
sc3 = mk_Score (<SouthAfrica>, 0, 2, 1, 2);
sc4 = mk_Score (<SaudiArabia>, 0, 1, 2, 1);
```

we have

```
iota x in set {sc1,sc2,sc3,sc4} & x.team = <France>  ≡  sc1
iota x in set {sc1,sc2,sc3,sc4} & x.points > 3       ≡  ⊥
iota x :  Score & x.points < x.won                   ≡  ⊥
```

Notice that the last example cannot be executed and that the last two expressions are undefined - in the former case because there is more than value satisfying the expression, and in the latter because no value satisfies the expression.

7.7 Set Expressions

Syntax: expression = ...
| set enumeration
| set comprehension
| set range expression
| ... ;

set enumeration = '{', [expression list], '}' ;

expression list = expression, { ',', expression } ;

set comprehension = '{', expression, '|', bind list,
['&', expression], '}' ;

set range expression = '{', expression, ',', '...', ',',
expression, '}' ;

Semantics: A *Set enumeration* has the form:

```
{e1, e2, e3, ..., en}
```

where `e1` up to `en` are general expressions. It constructs a set of the values of the enumerated expressions. The empty set must be written as `{}`.

The *set comprehension* expression has the form:

```
{e | mbd1, mbd2, ..., mbdn & P}
```

It constructs a set by evaluating the expression `e` on all the bindings for which the predicate `P` evaluates to `true`. A multiple binding can contain both set bindings and type bindings. Thus `mbdn` will look like `pat1 in set s1, pat2 : tp1, ...in set s2`, where `pati` is a pattern (normally simply an identifier), and `s1` and `s2` are sets constructed by expressions (whereas `tp1` is used to illustrate that type binds can also be used). Notice however that type binds cannot be executed by the interpreter.

The *set range expression* is a special case of a set comprehension. It has the form

```
{e1, ..., e2}
```

where `e1` and `e2` are numeric expressions. The set range expression denotes the set of integers from `e1` to `e2` inclusive. If `e2` is smaller than `e1` the set range expression denotes the empty set.

Examples: Using the values `Europe={<France>,<England>,<Denmark>,<Spain>}` and `GroupC = {sc1,sc2,sc3,sc4}` (where `sc1,...,sc4` are as defined in the preceding example) we have

```
{<France>, <Spain>} subset Europe            ≡  true
{<Brazil>, <Chile>, <England>}               ≡  false
      subset Europe
{<France>, <Spain>, "France"}                ≡  false
      subset Europe
{sc.team | sc in set GroupC                  ≡  {<France>,
      & sc.points > 2}                           <Denmark>}
{sc.team | sc in set GroupC                  ≡  {<SouthAfrica>,
      & sc.lost > sc.won }                       <SaudiArabia>}
{2.718,...,3.141}                            ≡  {3}
{3.141,...,2.718}                            ≡  {}
{1,...,5}                                    ≡  {1,2,3,4,5}
{ x | x:nat & x < 10 and x mod 2 = 0}        ≡  {0,2,4,6,8}
```

7.8 Sequence Expressions

Syntax: expression = ...
| sequence enumeration
| sequence comprehension
| subsequence
| ... ;

sequence enumeration = '[', [expression list], ']' ;

sequence comprehension = '[', expression, '|', set bind,
['&', expression], ']' ;

subsequence = expression,
'(', expression, ',', '...', ',',
expression, ')' ;

Semantics: A *sequence enumeration* has the form:

```
[e1, e2, ..., en]
```

where `e1` through `en` are general expressions. It constructs a sequence of the enumerated elements. The empty sequence must be written as `[]`.

A *sequence comprehension* has the form:

```
[e | pat in set S & P]
```

where the expression `e` will use the identifiers from the pattern `pat` (normally this pattern will simply be an identifier, but the only real requirement is that exactly one pattern identifier must be present in the pattern). `S` is a set of values (normally natural numbers). The bindings of the pattern identifier must be to some kind of numeric values which then are used to indicate the ordering of the elements in the resulting sequence. It constructs a sequence by evaluating the expression `e` on all the bindings for which the predicate `P` evaluates to `true`.

A *subsequence* of a sequence `l` is a sequence formed from consecutive elements of `l`; from index `n1` up to and including index `n2`. It has the form:

```
l(n1, ..., n2)
```

where `n1` and `n2` are positive integer expressions. If the lower bound `n1` is smaller than 1 (the first index in a non-empty sequence) the subsequence expression will start from the first element of the sequence. If the upper bound `n2` is larger than the length of the sequence (the largest index which can be used for a non-empty sequence) the subsequence expression will end at the last element of the sequence.

Examples: Given that `GroupA` is equal to the sequence

```
[ mk_Score(<Brazil>,2,0,1,6),
  mk_Score(<Norway>,1,2,0,5),
  mk_Score(<Morocco>,1,1,1,4),
  mk_Score(<Scotland>,0,1,2,1) ]
```

then:

```
[GroupA(i).team                    ≡  [<Brazil>,
| i in set inds GroupA                 <Norway>,
   & GroupA(i).won <> 0]               <Morocco>]
[GroupA(i)                         ≡  [mk_Score(<Scotland>,0,1,2,1)]
| i in set inds GroupA
   & GroupA(i).won = 0]
GroupA(1,...,2)                    ≡  [mk_Score(<Brazil>,2,0,1,6),
                                       mk_Score(<Norway>,1,2,0,5)]
[GroupA(i)                         ≡  []
| i in set inds GroupA
   & GroupA(i).points = 9]
```

7.9 Map Expressions

Syntax: expression = ...
| map enumeration
| map comprehension
| ... ;

map enumeration = '{', maplet, { ',', maplet }, '}'
| '{', '|->', '}' ;

maplet = expression, '|->', expression ;

map comprehension = '{', maplet, '|', bind list,
['&', expression], '}' ;

Semantics: A *map enumeration* has the form:

```
{d1 |-> r1, d2 |-> r2, ..., dn |-> rn}
```

where all the domain expressions `di` and range expressions `ri` are general expressions. The empty map must be written as `{|->}`.

A *map comprehension* has the form:

```
{ed |-> er | mbd1, ..., mbdn & P}
```

where constructs `mbd1, ..., mbdn` are multiple bindings of variables from the expressions `ed` and `er` to sets (or types). The *map comprehension* constructs a mapping by evaluating the expressions `ed` and `er` on all the possible bindings for which the predicate `P` evaluates to `true`.

Examples: Given that `GroupG` is equal to the map

```
{ <Romania> |-> mk_(2,1,0), <England> |-> mk_(2,0,1),
  <Colombia> |-> mk_(1,0,2), <Tunisia> |-> mk_(0,1,2) }
```

then:

```
{ t |-> let mk_(w,d,-) = GroupG(t)          ≡  {<Romania> |-> 7,
        in w * 3 + d                            <England> |-> 6,
| t in set dom GroupG}                          <Colombia> |-> 3,
                                                <Tunisia> |-> 1}
{ t |-> w * 3 + d                           ≡  {<Romania> |-> 7,
| t in set dom GroupG, w,d,l:nat                <England> |-> 6}
 & mk_(w,d,l) = GroupG(t)
 and w > l}
```

7.10 Tuple Constructor Expressions

Syntax: expression = ...
| tuple constructor
| ... ;

tuple constructor = 'mk_', '(', expression, ',', expression list, ')' ;

Semantics: The *tuple constructor expression* has the form:

```
mk_(e1, e2, ..., en)
```

where `ei` is a general expression. It can only be used by the equality and inequality operators.

Examples: Using the map `GroupG` defined in the preceding example, we have:

```
mk_(2,1,0) in set rng GroupG                   ≡ true
mk_("Romania",2,1,0) not in set rng GroupG     ≡ true
mk_(<Romania>,2,1,0) <> mk_("Romania",2,1,0)   ≡ true
```

7.11 Record Expressions

Syntax: expression = ...
| record constructor
| record modifier
| ... ;

record constructor = ‘mk_’, name, ‘(’, [expression list], ‘)’ ;

record modifier = ‘mu’, ‘(’, expression, ‘,’, record modification, { ‘,’, record modification } ‘)’ ;

record modification = identifier, ‘|->’, expression ;

Semantics: The *record constructor* has the form:

```
mk_T(e1, e2, ..., en)
```

where the type of the expressions `(e1, e2, ..., en)` matches the type of the corresponding entrances in the composite type `T`.

The *record modification* has the form:

```
mu (e, id1 |-> e1, id2 |-> e2, ..., idn |-> en)
```

where the evaluation of the expression `e` returns the record value to be modified. All the identifiers `idi` must be distinct named entrances in the record type of `e`.

Examples: If `sc` is the value `mk_Score(<France>,3,0,0,9)` then

```
mu(sc, drawn |-> sc.drawn + 1, points |-> sc.points + 1)
≡ mk_Score(<France>,3,1,0,10)
```

Further examples are demonstrated in the function `win`. This function takes two teams and a set of scores. From the set of scores it locates the scores corresponding to the given teams (`wsc` and `lsc` for the winning and losing team respectively), then updates these using the `mu` operator. The set of teams is then updated with the new scores replacing the original ones.

```
win : Team * Team * set of Score -> set of Score
win (wt,lt,gp) ==
  let wsc = iota sc in set gp & sc.team = wt,
      lsc = iota sc in set gp & sc.team = lt
  in let new_wsc = mu(wsc, won |-> wsc.won + 1,
                        points |-> wsc.points + 3),
         new_lsc = mu(lsc, lost |-> lsc.lost + 1)
     in (gp \ {wsc,lsc}) union {new_wsc, new_lsc}
pre forall sc1, sc2 in set gp &
          sc1 <> sc2 <=> sc1.team <> sc2.team
          and {wt,lt} subset {sc.team | sc in set gp}
```

7.12 Apply Expressions

Syntax: expression = . . .
| apply
| field select
| tuple select
| function type instantiation
| . . . ;

apply = expression, ‘(’, [expression list], ‘)’ ;

field select = expression, ‘.’, identifier ;

tuple select = expression, ‘.#’, numeral ;

function type instantiation = name, ‘[’, type, { ‘,’, type }, ‘]’ ;

Semantics: The *field select expression* can be used for records and it has already been explained in section 4.2.5 so no further explanation will be given here.

The *apply* is used for looking up in a map, indexing in a sequence, and finally for calling a function. In section 4.2.3 it has already been shown what it means to look up in a map. Similarly in section 4.2.2 it is illustrated how indexing in a sequence is performed.

In VDM-SL an operation can also be called here. This is not allowed in standard VDM-SL and because this kind of operation call can modify the state such usage should be done with care in complex expressions. Note however that such operation calls are not allowed to throw exceptions.

With such operation calls the order of evaluation can become important. Therefore the type checker will allow the user to enable or disable operation calls inside expressions.

The tuple select expression is used to extract a particular component from a tuple. The meaning of the expression is if `e` evaluates to some tuple `mk_(v1,...,vN)` and `M` is an integer in the range `{1,...,N}` then `e.#M` yields `vM`. If `M` lies outside `{1,...,N}` the expression is undefined.

The *function type instantiation* is used for instantiating polymorphic functions with the proper types. It has the form:

```
pf [ t1, ..., tn ]
```

where `pf` is the name of a polymorphic function, and `t1, ..., tn` are types. The resulting function uses the types `t1, ..., tn` instead of the variable type names given in the function definition.

Examples: Recall that `GroupA` is a sequence (page 52), `GroupG` is a map (page 53) and `selection_sort` is a function (page 41):

`GroupA(1)`	≡	`mk_Score(<Brazil>,2,0,1,6)`
`GroupG(<Romania>)`	≡	`mk_(2,1,0)`
`GroupG(<Romania>).#2`	≡	`1`
`selection_sort([3,2,9,1,3])`	≡	`[1,2,3,3,9]`

As an example of the use of polymorphic functions and function type instantiation, we use the example functions from section 6:

```
let emptyInt = empty_bag[int] in
  plus_bag[int](-1, emptyInt())
```

$\equiv$

```
{ -1 |-> 1 }
```

7.13 The Lambda Expression

Syntax: expression = ...
| lambda expression
| ... ;

lambda expression = 'lambda', type bind list, '&', expression ;

type bind list = type bind, { ',', type bind } ;

type bind = pattern, ':', type ;

Semantics: A *lambda expression* is of the form:

```
lambda pat1 : T1, ..., patn : Tn & e
```

where the `pati` are patterns, the `Ti` are type expressions, and `e` is the body expression. The scope of the pattern identifiers in the patterns `pati` is the body expression. A lambda expression cannot be polymorphic, but apart from that, it corresponds semantically to an explicit function definition as explained in section 6. A function defined by a lambda expression can be Curried by using a new lambda expression in the body of it in a nested way. When lambda expressions are bound to an identifier they can also define a recursive function.

Examples: An increment function can be defined by means of a lambda expression like:

```
Inc = lambda n : nat & n + 1
```

and an addition function can be Curried by:

```
Add = lambda a : nat & lambda b : nat & a + b
```

which will return a new lambda expression if it is applied to only one argument:

```
Add(5) ≡ lambda b : nat & 5 + b
```

Lambda expression can be useful when used in conjunction with higher-order functions. For instance using the function `set_filter` defined on page 40:

```
set_filter[nat](lambda n:nat & n mod 2 = 0)({1,...,10})
≡ {2,4,6,8,10}
```

7.14 Narrow Expressions

Syntax: expression = ...
| narrow expression
| ... ;

narrow expression = ‘narrow_’, ‘(’, expression, ‘,’, type, ‘)’ ;

Semantics: The *narrow expression* convert the given `expression` value into given `type`. Downcasting in class inheritance, and narrowing Union type are permit. However, conversions between unrelated types become type errors.

Examples: In following examples, there is no difference in the results of running the Test() and Test’(), But, there is a type error (DEF) in Test().

```
class A

types
public C1 :: a : nat;
public C2 :: b : nat;
public S = C1 | C2;

operations
public
Test: () ==> nat
Test() ==
```

```
  let s : S = mk_C1(1)
  in
    let c : C1 = s
    in
      return c.a;

public
Test': () ==> nat
Test'() ==
  let s : S = mk_C1(1)
  in
    let c : C1 = narrow_(s, C1)
    in
      return c.a;
end A
```

7.15 Is Expressions

Syntax: expression = ...
| general is expression
| ... ;

general is expression = is expression
| type judgement ;

is expression = 'is_', name, '(', expression, ')'
| is basic type, '(', expression, ')' ;

is basic type = 'is_', ('bool' | 'nat' | 'nat1' | 'int'
| 'rat' | 'real'
| 'char' | 'token') ;

type judgement = 'is_', '(', expression, ',', type, ')' ;

Semantics: The *is expression* can be used with values that are either basic or record values (tagged values belonging to some composite type). The is expression yields true if the given value belongs to the basic type indicated or if the value has the indicated tag. Otherwise it yields false.

A type judgement is a more general form which can be used for expressions whose types can not be statically determined. The expression `is_(e,t)` is equal to true if and only if `e` is of type `t`.

Examples: Using the record type `Score` defined on page 25 we have:

```
is_Score(mk_Score(<France>,3,0,0,9))   ≡ true
is_bool(mk_Score(<France>,3,0,0,9))    ≡ false
is_real(0)                             ≡ true
is_nat1(0)                             ≡ false
```

An example of a type judgement:

```
Domain : map nat to nat | seq of (nat*nat) -> set of nat
Domain(m) ==
  if is_(m, map nat to nat)
  then dom m
  else {d | mk_(d,-) in set elems m}
```

In addition there are examples on page 27.

7.16 Literals and Names

Syntax: expression = ...
| name
| old name
| symbolic literal
| ... ;

name = identifier, ['`', identifier] ;

name list = name, { ',', name } ;

old name = identifier, '~' ;

Semantics: *Names* and *old names* are used to access definitions of functions, operations, values and state components. A *name* has the form:

```
id1`id2
```

where `id1` and `id2` are simple identifiers. If a name consists of only one identifier, the identifier is defined within scope, i.e. it is defined either locally as a pattern identifier or variable, or globally within the current module as a function, operation, value or global variable. Otherwise, the identifier id1 indicates the module name where the construct is defined (see also section 14 and appendix B.)

An *old name* is used to access the old value of global variables in the post condition of an operation definition (see section 12) and in the post condition of specification statements (see section 13.14). It has the form:

```
id~
```

where id is a state component.

Symbolic literals are constant values of some basic type.

Examples: *Names* and *symbolic literals* are used throughout all examples in this document (see appendix B.2).

For an example of the use of *old names*, consider the state defined as:

```
state sigma of
  numbers : seq of nat
  index   : nat
inv   mk_sigma(numbers, index) == index not in set elems numbers
init  s == s = mk_sigma([], 1)
end
```

We can define an operation that increases the variable index in an implicit manner:

```
IncIndex()
ext wr index : nat
post index = index~ + 1
```

The operation IncIndex manipulates the variable index, indicated with the ext wr clause. In the post condition, the new value of index is equal to the old value of index plus 1. (See more about operations in section 12).

For a simple example of module names, suppose that a function called build_rel is defined (and exported) in a module called CGRel as follows:

```
types

  Cg = <A> | <B> | <C> | <D> | <E> | <F> |
       <G> | <H> | <J> | <K> | <L> | <S>;
  CompatRel = map Cg to set of Cg

functions

  build_rel : set of (Cg * Cg) -> CompatRel
  build_rel (s) == {|->}
```

In another module we can access this function by first importing the module `CGRel` then by using the following call

```
CGRel`build_rel(mk_(<A>, <B>))
```

7.17 The Undefined Expression

Syntax: expression = ...
| undefined expression ;

undefined expression = ‘undefined’ ;

Semantics: The *undefined expression* is used to state explicitly that the result of an expression is undefined. This could for instance be used if it has not been decided what the result of evaluating the else-branch of an if-then-else expression should be. When an *undefined expression* is evaluated the interpreter will terminate the execution and report that an undefined expression was evaluated.

Pragmatically use of undefined expressions differs from pre-conditions: use of a pre-condition means it is the caller's responsibility to ensure that the pre-condition is satisfied when the function is called; if an undefined expression is used it is the called function's responsibility to deal with error handling.

Examples: We can check that the type invariant holds before building `Score` values:

```
build_score : Team * nat * nat * nat * nat -> Score
build_score (t,w,d,l,p) ==
```

```
if 3 * w + d = p
then mk_Score(t,w,d,l,p)
else undefined
```

7.18 The Precondition Expression

Syntax: expression = ...
| precondition expression ;

precondition expression = 'pre_', '(', expression,
[{ ',', expression }], ')' ;

Semantics: Assuming `e` is of function type the expression `pre_(e,e1,...,en)` is true if and only if the pre-condition of `e` is true for arguments `e1,...,em` where `m` is the arity of the pre-condition of `e`. If `e` is not a function or `m` > `n` then the result is `true`. If `e` has no pre-condition then the expression equals true.

Examples: Consider the functions `f` and `g` defined below

```
f : nat * nat -> nat
f(m,n) == m div n
pre n <> 0;

g (n: nat) sqrt:nat
pre n >= 0
post sqrt * sqrt <= n and
     (sqrt+1) * (sqrt+1) > n
```

Then the expression

```
pre_(let h in set {f,g, lambda mk_(x,y):nat * nat & x div y}
     in h, 1,0,-1)
```

is equal to

- false if `h` is bound to `f` since this equates to `pre_f(1,0)`;
- true if `h` is bound to `g` since this equates to `pre_g(1)`;

- true if `h` is bound to `lambda mk_(x,y):nat * nat & x div y` since there is no pre-condition defined for this function.

Note that however `h` is bound, the last argument (`-1`) is never used.

8 Patterns

Syntax: pattern bind = pattern | bind ;

pattern = pattern identifier
| match value
| set enum pattern
| set union pattern
| seq enum pattern
| seq conc pattern
| map enumeration pattern
| map muinon pattern
| tuple pattern
| record pattern ;

pattern identifier = identifier | '-' ;

match value = symbolic literal
| '(', expression, ')' ;

set enum pattern = '{', [pattern list], '}' ;

set union pattern = pattern, 'union', pattern ;

seq enum pattern = '[', [pattern list], ']' ;

seq conc pattern = pattern, '^', pattern ;

map enumeration pattern = '{', [maplet pattern list], '}' ;

maplet pattern list = maplet pattern, { ',', maplet pattern } ;

maplet pattern = pattern, '|->', pattern ;

map muinon pattern = pattern, 'munion', pattern ;

tuple pattern = 'mk_(', pattern, ',', pattern list, ')' ;

record pattern = 'mk_', name, '(', [pattern list], ')' ;

pattern list = pattern, { ',', pattern } ;

Semantics: A pattern is always used in a context where it is matched to a value of a particular type. Matching consists of checking that the pattern can be matched to the value, and binding any pattern identifiers in the pattern to the corresponding values, i.e. making the identifiers denote those values throughout their scope. In some cases where a pattern can be used, a bind can be used as well (see next section). If a bind is used it simply means that additional information (a type or a set expression) is used to constrain the possible values which can match the given pattern.

Matching is defined as follows

1. A *pattern identifier* fits any type and can be matched to any value. If it is an identifier, that identifier is bound to the value; if it is the don't-care symbol '-', no binding occurs.
2. A *match value* can only be matched against the value of itself; no binding occurs. If a match value is not a literal like e.g. `7` or `<RED>` it must be an expression enclosed in parentheses in order to discriminate it to a pattern identifier.
3. A *set enumeration pattern* fits only set values. The patterns are matched to distinct elements of a set; all elements must be matched.
4. A *set union pattern* fits only set values. The two patterns are matched to a partition of two subsets of a set. In the Toolbox the two subsets will always be chosen such that they are non-empty and disjoint.
5. A *sequence enumeration pattern* fits only sequence values. Each pattern is matched against its corresponding element in the sequence value; the length of the sequence value and the number of patterns must be equal.
6. A *sequence concatenation pattern* fits only sequence values. The two patterns are matched against two subsequences which together can be concatenated to form the original sequence value. In the Toolbox the two subsequences will always be chosen so that they are non-empty.
7. A *map enumeration pattern* fits only map values.
8. A *maplet pattern list* are matched to distinct elements of a map; all elements must be matched.

9. A *map munion pattern* fits only map values. The two patterns are matched to a partition of two sub maps of a map. In the VDM interpreters the two sub maps will always be chosen such that they are non-empty and disjoint.
10. A *tuple pattern* fits only tuples with the same number of elements. Each of the patterns are matched against the corresponding element in the tuple value.
11. A *record pattern* fits only record values with the same tag. Each of the patterns are matched against the field of the record value. All the fields of the record must be matched.

Examples: The simplest kind of pattern is the pattern identifier. An example of this is given in the following let expression:

```
let top = GroupA(1)
in top.sc
```

Here the identifier `top` is bound to the head of the sequence `GroupA` and the identifier may then be used in the body of the let expression.

In the following examples we use match values:

```
let a = <France>
in cases GroupA(1).team:
          <Brazil> -> "Brazil are winners",
          (a)      -> "France are winners",
          others   -> "Neither France nor Brazil are winners"
   end;
```

Match values can only match against their own values, so here if the team at the head of `GroupA` is `<Brazil>` then the first clause is matched; if the team at the head of `GroupA` is `<France>` then the second clause is matched. Otherwise the `others` clause is matched. Note here that the use of brackets around `a` forces `a` to be considered as a match value.

Set enumerations match patterns to elements of a set. For instance in

```
let {sc1, sc2, sc3, sc4} = elems GroupA
in sc1.points + sc2.points + sc3.points + sc4.points;
```

the identifiers `sc1`, `sc2`, `sc3` and `sc4` are bound to the four elements of `GroupA`. Note that the choice of binding is loose - for instance `sc1` may be bound to [any] element of elems GroupA. In this case if elems GroupA does not contain precisely four elements, then the expression is not well-formed.

A set union pattern can be used to decompose a set for recursive function calls. An example of this is the function set2seq which converts a set into a sequence (with arbitrary order):

```
set2seq[@elem] : set of @elem -> seq of @elem
set2seq(s) ==
  cases s:
    {} -> [],
    {x} -> [x],
    s1 union s2 -> (set2seq[@elem](s1))^(set2seq[@elem](s2))
  end
```

In the third cases alternative we see the use of a set union pattern. This binds s1 and s2 to arbitrary subsets of s such that they partition s. The Toolbox interpreter always ensures a disjoint partition.

Sequence enumeration patterns can be used to extract specific elements from a sequence. An example of this is the function promoted which extracts the first two elements of a seqnce of scores and returns the corresponding pair of teams:

```
promoted : seq of Score -> Team * Team
promoted([sc1,sc2]^-) == mk_(sc1.team,sc2.team);
```

Here sc1 is bound to the head of the argument sequence, and sc2 is bound to the second element of the sequence. If promoted is called with a sequence with fewer than two elements then a runtime error occurs. Note that as we are not interested in the remaining elements of the list we use a don't care pattern for the remainder.

The preceding example also demonstrated the use of sequence concatenation patterns. Another example of this is the function quicksort which implements a standard quicksort algorithm:

```
quicksort : seq of nat -> seq of nat
quicksort (l) ==
  cases l:
    [] -> [],
    [x] -> [x],
    [x,y] -> if x < y then [x,y] else [y,x],
```

```
    -^[x]^- -> quicksort ([y | y in set elems l & y < x]) ^
               [x] ^ quicksort ([y | y in set elems l & y > x])
  end
```

Here, in the second cases clause a sequence concatenation pattern is used to decompose `l` into an arbitrary pivot element and two subsequences. The pivot is used to partition the list into those values less than the pivot and those values greater, and these two partitions are recursively sorted.

Maplet pattern match patterns to elements of a maplet.

```
let {a |-> b} = {1 |-> 2} in mk_(a,b) = mk_(1,2)
```

Maplet pattern list match patterns to elements of each maplet in a map.

```
let {1 |-> a,a |-> b,b |-> c} = {1 |-> 4,2 |-> 3,4 |-> 2} in
c = 3
```

Map munion pattern can be used to decompose a map for recursive function calls. Following `map2seq` function converts a map to a seq of maplet.

```
map2seq[@T1, @T2] : map @T1 to @T2 -> seq of (map @T1 to @T2)
map2seq(m) ==
  cases m:
    ({|->}) -> [],
    {- |-> -} -> [m],
    m1 munion m2 -> map2seq[@T1, @T2] (m1) ^  map2seq[@T1, @T2] (m2)
  end;
```

Here, in the third cases clause a map munion pattern is used to decompose `m` into two maps.

Tuple patterns can be used to bind tuple components to identifiers. For instance since the function `promoted` defined above returns a pair, the following value definition binds the winning team of `GroupA` to the identifier `Awinner`:

```
  values

    mk_(Awinner,-) = promoted(GroupA);
```

Record patterns are useful when several fields of a record are used in the same expression. For instance the following expression constructs a map from team names to points score:

```
{ t |-> w * 3 + 1 | mk_Score(t,w,1,-,-) in set elems GroupA}
```

The function `print_Expr` on page 45 also gives several examples of record patterns.

9 Bindings

Syntax: bind = set bind | type bind ;

set bind = pattern, 'in set', expression ;

type bind = pattern, ':', type ;

bind list = multiple bind, { ',', multiple bind } ;

multiple bind = multiple set bind
| multiple type bind ;

multiple set bind = pattern list, 'in set', expression ;

multiple type bind = pattern list, ':', type ;

Semantics: A *bind* matches a pattern to a value. In a *set bind* the value is chosen from the set defined by the set expression of the bind. In a *type bind* the value is chosen from the type defined by the type expression. *Multiple bind* is the same as *bind* except that several patterns are bound to the same set or type. Notice that type binds **cannot** be executed by the interpreter. This would require the interpreter to search through infinite domains like the natural numbers.

Examples: Bindings are mainly used in quantified expressions and comprehensions which can be seen from these examples:

```
forall i, j in set inds list & i < j => list(i) <= list(j)

{ y | y in set S & y > 2 }

{ y | y: nat & y > 3 }

occurs : seq1 of char * seq1 of char -> bool
occurs (substr,str) ==
```

```
exists i,j in set inds str & substr = str(i,...,j);
```

10 Value (Constant) Definitions

VDM-SL supports the definition of constant values. A value definition corresponds to a constant definition in traditional programming languages.

Syntax: value definitions = ‘values’, [value definition, { ‘;’, value definition }, [‘;’]] ;

value definition = pattern, [‘:’, type], ‘=’, expression ;

Semantics: The value definition has the form:

```
values
  pat1 = e1;
 ...
  patn = en
```

The global values (defined in a value definition) can be referenced at all levels in a VDM-SL specification. However, in order to be able to execute a specification these values must be defined before they are used in the sequence of value definitions. This “declaration before use” principle is only used by the interpreter for value definitions. Thus for instance functions can be used before they are declared. In standard VDM-SL there are not any restrictions on the order of the definitions at all. It is possible to provide a type restriction as well, and this can be useful in order to obtain more exact type information.

Examples: The example below, taken from [3] assigns token values to identifiers `p1` and `eid2`, an `Expert` record value to `e3` and an `Alarm` record value to `a1`.

```
types

Period = token;
```

```
ExpertId = token;
Expert :: expertid : ExpertId
          quali : set of Qualification
inv ex == ex.quali <> {};
Qualification = <Elec> | <Mech> | <Bio> | <Chem>;
Alarm :: alarmtext : seq of char
         quali : Qualification

values

 p1: Period = mk_token("Monday day");
 eid2 : ExpertId = mk_token(145);
 e3 : Expert = mk_Expert(eid2,  <Mech>, <Chem> );
 a1 : Alarm = mk_Alarm("CO2 detected", <Chem>)
```

As this example shows, a value can depend on other values which are defined previous to itself. A top-level specification can consist of specifications from a number of files or modules (see section 14). It is good practice not to let a value depend on values defined in other modules as the ordering is important.

11 The State Definition

If global variables are desired in a specification, it is possible to make a state definition. The components of the state definition can be considered the collection of global variables which can be referenced inside operations. A state in a module is initialised before any of the operation definitions (using that state) in a module can be used by the interpreter.

Syntax: state definition = 'state', identifier, 'of', field list,
[invariant], [initialisation], 'end', [';'] ;

invariant = 'inv', invariant initial function ;

initialisation = 'init', invariant initial function ;

invariant initial function = pattern, '==', expression ;

Semantics: The state definition has the form:

```
state ident of
  id1 : type1
  ...
  idn : typen
inv   pat1 == inv
init  pat2 == init
end
```

A state identifier `idn` is declared of a specific type `typen`. The invariant `inv` is a boolean expression denoting a property which must hold for the state `ident` at all times. `init` denotes a condition which must hold initially. It should be noticed that in order to use the interpreter, it is necessary to have an initialisation predicate (if any of the operations using the state are to be executed). In addition the body of this initialisation predicate must be a binary equality expression with the name (which also must be used as the pattern) of the entire state on the left-hand side of the equality and the right-hand side must evaluate to a record value of the correct type. This enables the interpreter to evaluate the `init` condition. A simple example of an initialisation predicate is shown below:

```
state St of
  x:nat
  y:nat
  l:seq1 of nat
init s == s = mk_St(0,0,[1])
end
```

In the specification of both the invariant and the initial value the state must be manipulated as a whole, and this is done by referring to it as a record tagged with the state name (see the example). When a field in the state is manipulated in some operation, the field must however be referenced to directly by the field name without pre-fixing it with the state name.

Examples: In the following example we create one state variable:

```
types

GroupName = <A> | <B> | <C> | <D> | <E> | <F> | <G> | <H>
```

```
state GroupPhase of
  gps : map GroupName to set of Score
inv mk_GroupPhase(gps) ==
  forall gp in set rng gps &
    (card gp = 4 and
     forall sc in set gp & sc.won + sc.lost + sc.drawn <= 3)
init gp ==
  gp = mk_GroupPhase ( <A> |->
                              init_sc (<Brazil>, <Norway>,
                                  <Morocco>, <Scotland>),
                        ...)
end

functions

init_sc : set of Team -> set of Score
init_sc (ts) ==
  { mk_Score (t,0,0,0,0) | t in set ts }
```

In the invariant we state that each group has four teams, and no team plays more than three games. Initially no team has played any games.

12 Operation Definitions

Operations have already been mentioned in section 5. The general form is described here.

Syntax: operation definitions = ‘operations’, [operation definition, { ‘;’, operation definition }, [‘;’]] ;

operation definition = explicit operation definition
| implicit operation definition
| extended explicit operation definition ;

explicit operation definition = identifier, ‘:’, operation type,
identifier, parameters,
‘==’,
operation body,

[‘pre’, expression],
[‘post’, expression] ;

implicit operation definition = identifier, parameter types,
[identifier type pair list],
implicit operation body ;

implicit operation body = [externals],
[‘pre’, expression],
‘post’, expression,
[exceptions] ;

extended explicit operation definition = identifier,
parameter types,
[identifier type pair list],
‘==’, operation body,
[externals],
[‘pre’, expression],
[‘post’, expression],
[exceptions] ;

operation type = discretionary type, ‘==>’, discretionary type ;

discretionary type = type | ‘()’ ;

parameters = ‘(’, [pattern list], ‘)’ ;

pattern list = pattern, { ‘,’, pattern } ;

operation body = statement
| ‘is not yet specified’ ;

externals = ‘ext’, var information, { var information } ;

var information = mode, name list, [‘:’, type] ;

mode = ‘rd’ | ‘wr’ ;

name list = identifier, { ‘,’, identifier } ;

exceptions = ‘errs’, error list ;

error list = error, { error } ;

error = identifier, ':', expression, '->', expression ;

Semantics: The following example of an explicit operation updates the state GroupPhase when one team beats another.

```
Win : Team * Team ==> ()
Win (wt,lt) ==
  let gp in set dom gps be st
       {wt,lt} subset {sc.team | sc in set gps(gp)}
  in gps := gps ++ { gp |->
                     {if sc.team = wt
                      then mu(sc, won |-> sc.won + 1,
                                  points |-> sc.points + 3)
                      else if sc.team = lt
                      then mu(sc, lost |-> sc.lost + 1)
                      else sc
                 | sc in set gps(gp)}}
  pre exists gp in set dom gps &
        {wt,lt} subset {sc.team | sc in set gps(gp)};
```

An explicit operation consists of a statement (or several composed using a block statement), as described in section 13. The statement may access any state variables it wishes, reading and writing to them as it sees fit.

An implicit operation is specified using an optional pre-condition, and a mandatory post-condition. For example we could specify the Win operation implicitly:

```
Win (wt,lt: Team)
ext wr  gps : map GroupName to set of Score
pre  exists gp in set dom gps &
        {wt,lt} subset {sc.team | sc in set gps(gp)}
post  exists gp in set dom gps &
       {wt,lt} subset {sc.team | sc in set gps(gp)}
       and gps = gps~ ++
                    { gp |->
                      {if sc.team = wt
                       then mu(sc, won |-> sc.won + 1,
                                   points |-> sc.points + 3)
                       else if sc.team = lt
                       then mu(sc, lost |-> sc.lost + 1)
                       else sc
                  | sc in set gps(gp)}};
```

The externals field lists the state variables that the operation will manipulate. The state variables listed after the reserved word rd can only be read whereas the operation can both read and write the variables listed after wr.

For these pre- and post-conditions the interpreter also creates new functions as with the pre- and post-conditions of operation definitions. However, if a specification contains a global state, the state is also part of the newly created functions. Thus, functions with the following signatures are created for operations with pre- and/or post-conditions[16]:

```
pre_Op : InType * State +> bool

post_Op : InType * OutType * State * State +> bool
```

with the following exceptions:

- If the operation does not take any arguments, the `InType` part of the signature is left out in both the `pre_Op` and `post_Op` signatures.
- If the operation does not return a value, the `OutType` part is left out in the `post_Op` signature.
- If the specification does not define a state, the `State` part(s) of both signatures are left out.

In the `post_Op` signature, the first `State` part is for the old state, whereas the second `State` part is for the state after the operation call.

For instance, consider the following specifications:

[16] However, you should remember that these pre and post condition predicates for an operation are simply boolean functions and the state components are thus not changed by calling such a predicate.

```
module A

definitions

state St of
  n : nat
end

operations

Op1 (a : nat) b :nat
pre a > 0
post b = 2 * a;

Op2 () b : nat
post b = 2;

Op3 ()
post true

end A
```

```
module B

definitions

operations

Op1 (a : nat) b : nat
pre a > 0
post b = 2 * a;

Op2 () b : nat
post b = 2;

Op3 ()
post true

end B
```

For **module A** we could then quote the pre and post conditions defined in this specification as illustrated below

Quote expression	Explanation
`pre_Op1(1,mk_St(2))`	`a` bound to 1 in state `St` with `n` bound to 2
`post_Op1(1,2,mk_St(1), mk_St(2))`	`a` bound to 1, `b` bound to 2, state before with `n` bound to 1, state after with `n` bound to 2
`post_Op2(2,mk_St(1), mk_St(2))`	`b` bound to 2, state before with `n` bound to 1, state after with `n` bound to 2
`post_Op3(mk_St(1), mk_St(2))`	state before with `n` bound to 1, state after with `n` bound to 2

For **module B** we can quote the pre and post conditions defined in this specification as illustrated below

Quote expression	Explanation
`pre_Op1(1)`	`a` bound to 1
`post_Op1(1,2)`	`a` bound to 1, `b` bound to 2
`post_Op2(2)`	`b` bound to 2
`post_Op3()`	No binding at all

The exceptions clause can be used to describe how an operation should deal with error situations. The rationale for having the exception clause is to give the user the ability to separate the exceptional cases from the normal cases. The specification using exceptions does not give any commitment as to how exceptions are to be signalled, but it gives the means to show under which circumstances an error situation can occur and what the consequences are for the result of calling the operation.

The exception clause has the form:

```
errs COND1: c1 -> r1
     ...
     CONDn: cn -> rn
```

The condition names `COND1`, ..., `CONDn` are identifiers which describe the kind of error which can be raised[17]. The condition expressions `c1`, ..., `cn` can be considered as pre-conditions for the different kinds of errors. Thus, in these expressions the identifiers from the arguments list and the variables from the externals list can be used (they have the same scope as the pre-condition). The result expressions `r1`, ..., `rn` can correspondingly be considered as post-conditions for the different kinds of errors. In these expressions the result identifier and old values of global variables (which can be written to) can also be used. Thus, the scope corresponds to the scope of the post-condition.

Superficially there appears to be some redundancy between exceptions and pre-conditions here. However there is a conceptual distinction between them which dictates which should be used and when. The pre-condition specifies what callers to the operation must ensure for correct behaviour; the exception clauses indicate that the operation being specified takes responsibility for error handling when an exception condition is satisfied. Hence normally exception clauses and pre-conditions do not overlap.

The next example of an operation uses the following state definition:

[17] Notice that these names are purely of mnemonic value, i.e. semantically they are not important.

```
state qsys of
  q : Queue
end
```

This example shows how exceptions with an implicit definition can be used:

```
DEQUEUE() e: [Elem]
ext wr q : Queue
post q~ = [e] ^ q
errs QUEUE_EMPTY: q = [] -> q = q~ and e = nil
```

This is a dequeue operation which uses a global variable `q` of type `Queue` to get an element `e` of type `Elem` out of the queue. The exceptional case here is that the queue in which the exception clause specifies how the operation should behave is empty.

Note that the Toolbox creates a function here:

```
post_DEQUEUE: [Elem] * qsys * qsys +> bool
```

13 Statements

In this section the different kind of statements will be described one by one. Each of them will be described by means of:

- A syntax description in BNF.
- An informal semantics description.
- An example illustrating its usage.

13.1 Let Statements

Syntax: statement = let statement
| let be statement
| . . . ;

let statement = 'let', local definition, { ',', local definition },
'in', statement ;

let be statement = 'let', bind, ['be', 'st', expression], 'in', statement ;

local definition = value definition
| function definition ;

value definition = pattern, [':', type], '=', expression ;

where the "function definition" component is described in section 6.

Semantics: The *let statement* and the *let-be-such-that statement* are similar to the corresponding *let* and *let-be-such-that expressions* except that the *in* part is a statement instead of an expression. Thus it can be explained as follows:

A simple *let statement* has the form:

```
let p1 = e1, ..., pn = en in s
```

where `p1, ..., pn` are patterns, `e1, ..., en` are expressions which match the corresponding patterns `pi`, and `s` is a statement, of any type, involving the pattern identifiers of `p1, ..., pn`. It denotes the evaluation of the statement `s` in the context in which the patterns `p1, ..., pn` are matched against the corresponding expressions `e1, ..., en`.

More advanced let statements can also be made by using local function definitions. The semantics of doing that is simply that the scope of such locally defined functions is restricted to the body of the let statement.

In VDM-SL the collection of definitions may be mutually recursive. However, this is not supported by the interpreter in VDM-SL. Furthermore, the definitions must be ordered such that all constructs are defined before they are used.

A *let-be-such-that statement* has the form

```
let b be st e in s
```

where `b` is a binding of a pattern to a set value (or a type), `e` is a boolean expression, and `s` is a statement, involving the pattern identifiers of the pattern in `b`. The `be st e` part is optional. The expression denotes the evaluation of the statement `s` in the context where the pattern from `b` has been matched against an element in the set (or type) from b[18]. If the be st expression e is present, only such bindings where e evaluates to true in the matching context are used.

Examples: An example of a let be st statement is provided in the operation GroupWinner which returns the winning team in a given group:

```
GroupWinner : GroupName ==> Team
GroupWinner (gp) ==
  let sc in set gps(gp) be st
     forall sc' in set gps(gp) \ {sc} &
        (sc.points > sc'.points) or
        (sc.points = sc'.points and sc.won > sc'.won)
  in return sc.team
```

The companion operation GroupRunnerUp gives an example of a simple let statement as well:

```
GroupRunnerUp_expl : GroupName ==> Team
GroupRunnerUp_expl (gp) ==
  def t = GroupWinner(gp)
  in let sct = iota sc in set gps(gp) & sc.team = t
    in
       let sc in set gps(gp) \ {sct} be st
         forall sc' in set gps(gp) \ {sc,sct} &
           (sc.points > sc'.points) or
           (sc.points = sc'.points and sc.won > sc'.won)
       in return sc.team
```

Note the use of the def statement (section 13.2) here; this is used rather than a let statement since the right-hand side is an operation call, and therefore is not an expression.

13.2 The Define Statement

Syntax: statement = …
| def statement
| … ;

[18]Remember that only the set bindings can be executed by means of the interpreter.

def statement = 'def', equals definition,
{ ';', equals definition },[';'], 'in',
statement ;

equals definition = pattern bind, '=', expression ;

Semantics: A *define statement* has the form:

```
def pb1 = e1;
    ...
    pbn = en
in
  s
```

The *define statement* corresponds to a *define expression* except that it is also allowed to use operation calls on the right-hand sides. Thus, operations that change the state can also be used here, and if there are more than one definition they are evaluated in the order in which they are presented. It denotes the evaluation of the statement `s` in the context in which the patterns (or binds) `pb1, ..., pbn` are matched against the values returned by the corresponding expressions or operation calls `e1, ..., en`[19].

Examples: Given the following sequences:

```
secondRoundWinners = [<A>,<B>,<C>,<D>,<E>,<F>,<G>,<H>];
secondRoundRunnersUp = [<B>,<A>,<D>,<C>,<F>,<E>,<H>,<G>]
```

The operation `SecondRound`returns the sequence of pairs representing the second round games gives an example of a def statement:

```
SecondRound : () ==> seq of (Team * Team)
SecondRound () ==
def winners = { gp |-> GroupWinner(gp) | gp in set dom gps };
    runners_up = { gp |-> GroupRunnerUp(gp) | gp in set dom gps}
in return ([mk_(winners(secondRoundWinners(i)),
                runners_up(secondRoundRunnersUp(i)))
          | i in set {1,...,8}])
```

[19]If binds are used it simply means that the values which can match the pattern are further constrained by the type or set expression as it is explained in section 8.

13.3 The Block Statement

Syntax: statement = ...
| block statement
| ... ;

block statement = '(', { dcl statement },
statement, { ';', statement }, [';'], ')' ;

dcl statement = 'dcl', assignment definition,
{ ',', assignment definition }, ';' ;

assignment definition = identifier, ':', type, [':=', expression] ;

Semantics: The *block statement* corresponds to block statements from traditional high-level programming languages. It enables the use of locally defined variables (by means of the declare statement) which can be modified inside the body of the block statement. It simply denotes the ordered execution of what the individual statements prescribe. The first statement in the sequence that returns a value causes the evaluation of the sequence statement to terminate. This value is returned as the value of the block statement. If none of the statements in the block returns a value, the evaluation of the block statement is terminated when the last statement in the block has been evaluated. When the block statement is left the values of the local variables are discharged. Thus, the scope of these variables is simply inside the block statement.

Examples: In the context of state definition

```
state St of
  x:nat
  y:nat
  l:seq1 of nat
end
```

the operation `Swap` uses a block statement to swap the values of variables `x` and `y`:

```
Swap : () ==> ()
Swap () ==
  (dcl temp: nat := x;
```

```
 x := y;
 y := temp
)
```

13.4 The Assignment Statement

Syntax: statement = ...
| assign statement
| ... ;

assign statement = state designator, ':=', expression ;

state designator = name
| field reference
| map or sequence reference ;

field reference = state designator, '.', identifier ;

map or sequence reference = state designator, '(', expression, ')' ;

multiple assign statement = 'atomic', '(' assign statement, ';',
assign statement,
{ ';', assign statement } ')' ;

Semantics: The *assignment statement* corresponds to a generalisation of assignment statements from traditional high level programming languages. It is used to change the value of the global or local state. Thus, the assignment statement has side-effects on the state. However, in order to be able to simply change a part of the state, the left-hand side of the assignment can be a state designator. A state designator is either simply the name of a global variable, a reference to a field of a variable, a map reference of a variable, or a sequence reference of a variable. In this way it is possible to change the value of a small component of the state. For example, if a state component is a map, it is possible to change a single entry in the map.

An assignment statement has the form:

```
sd := ec
```

where `sd` is a state designator, and `ec` is either an expression or a call of an operation. The assignment statement denotes the change to the given state component described at the right-hand side (expression or operation call). If the right-hand side is a state changing operation then that operation is executed (with the corresponding side effect) before the assignment is made.

Multiple assignment is also possible. This has the form:

```
atomic (sd1 := ec1;
        ...;
        sdN := ecN
        )
```

All of the expressions or operation calls on the right hand sides are executed or evaluated, and then the results are bound to the corresponding state designators. The right-hand sides are executed atomically with respect to invariant evaluation.

Examples: The operation in the previous example (`Swap`) illustrated normal assignment. The operation `Win_sd`, a refinement of `Win` on page 75 illustrates the use of state designators to assign to a specific map key:

```
Win_sd : Team * Team ==> ()
Win_sd (wt,lt) ==
  let gp in set dom gps be st
      {wt,lt} subset {sc.team | sc in set gps(gp)}
  in gps(gp) := { if sc.team = wt
                  then mu(sc, won |-> sc.won + 1,
                              points |-> sc.points + 3)
                  else if sc.team = lt
                  then mu(sc, lost |-> sc.lost + 1)
                  else sc
                | sc in set gps(gp)}
pre exists gp in set dom gps &
                {wt,lt} subset {sc.team | sc in set gps(gp)}
```

The operation `SelectionSort` is a state based version of the function `selection_sort` on page 41. It demonstrates the use of state designators to modify the contents of a specific sequence index, using the state `St` defined on page 83.

```
functions

min_index : seq1 of nat -> nat
min_index(l) ==
if len l = 1 then 1
else let mi = min_index(tl l)
   in if l(mi+1) < hd l
     then mi+1
     else 1

operations

SelectionSort : nat ==> ()
SelectionSort (i) ==
  if i < len l
  then (dcl temp: nat;
        dcl mi : nat := min_index(l(i,...,len l)) + i - 1;
        temp := l(mi);
        l(mi) := l(i);
        l(i) := temp;
        SelectionSort(i+1)
       );
```

13.5 Conditional Statements

Syntax: statement = ...
| if statement
| cases statement
| ... ;

if statement = 'if', expression, 'then', statement,
{ elseif statement }, ['else', statement] ;

elseif statement = 'elseif', expression, 'then', statement ;

cases statement = 'cases', expression, ':',
cases statement alternatives,
[',', others statement], 'end' ;

cases statement alternatives = cases statement alternative,
{ ‘,’, cases statement alternative } ;

cases statement alternative = pattern list, ‘->’, statement ;

others statement = ‘others’, ‘->’, statement ;

Semantics: The semantics of the *if statement* corresponds to the *if expression* described in section 7.4 except for the alternatives which are statements (and that the else part is optional)[20].

The semantics for the *cases statement* corresponds to the *cases expression* described in section 7.4 except for the alternatives which are statements.

Examples: Assuming functions `clear_winner` and `winner_by_more_wins` and operation `RandomElement` with the following signatures:

```
clear_winner : set of Score -> bool
winner_by_more_wins : set of Score -> bool
RandomElement : set of Team ==> Team
```

then the operation `GroupWinner_if` demonstrates the use of a nested if statement (the iota expression is presented on page 48):

```
GroupWinner_if : GroupName ==> Team
GroupWinner_if (gp) ==
  if clear_winner(gps(gp))
   -- return unique score in gps(gp) which has more points
   -- than any other score
  then return ((iota sc in set gps(gp) &
                  forall sc' in set gps(gp) \ {sc} &
                    sc.points > sc'.points).team)
  else if winner_by_more_wins(gps(gp))
   -- return unique score in gps(gp) with maximal points
   -- & has won more than other scores with maximal points
  then return ((iota sc in set gps(gp) &
             forall sc' in set gps(gp) f {sc} &
               (sc.points > sc'.points) or
               (sc.points = sc'.points and
                sc.won > sc'.won)).team)
```

[20]If the else part is omitted semantically it is like using else skip.

```
  -- no outright winner, so choose random score
  -- from joint top scores
 else RandomElement ( {sc.team | sc in set gps(gp) &
                        forall sc' in set gps(gp) &
                         sc'.points <= sc.points} );
```

Alternatively, we could use a cases statement with match value patterns for this operation:

```
GroupWinner_cases : GroupName ==> Team
GroupWinner_cases (gp) ==
  cases true:
    (clear_winner(gps(gp))) ->
         return ((iota sc in set gps(gp) &
                   forall sc' in set gps(gp) \ {sc} &
                    sc.points > sc'.points).team),

    (winner_by_more_wins(gps(gp))) ->
         return ((iota sc in set gps(gp) &
                   forall sc' in set gps(gp) \ {sc} &
                     (sc.points > sc'.points) or
                     (sc.points = sc'.points and
                        sc.won > sc'.won)).team),

    others -> RandomElement ( {sc.team | sc in set gps(gp) &
                                forall sc' in set gps(gp) &
                                 sc'.points <= sc.points} )
  end
```

13.6 For-Loop Statements

Syntax: statement = ...
| sequence for loop
| set for loop
| index for loop
| ... ;

sequence for loop = 'for', pattern bind, 'in', ['reverse'], expression, 'do', statement ;

set for loop = 'for', 'all', pattern, 'in set', expression,
'do', statement ;

index for loop = 'for', identifier, '=', expression, 'to', expression,
['by', expression], 'do', statement ;

Semantics: There are three kinds of *for-loop statements*. The for-loop using an index is known from most high-level programming languages. In addition, there are two for-loops for traversing sets and sequences. These are especially useful if access to all elements from a set (or sequence) is needed one by one.

An *index for-loop statement* has the form:

```
for id = e1 to e2 by e3 do
s
```

where `id` is an identifier, `e1` and `e2` are integer expressions indicating the lower and upper bounds for the loop, `e3` is an integer expression indicating the step size, and `s` is a statement where the identifier `id` can be used. It denotes the evaluation of the statement `s` as a sequence statement where the current context is extended with a binding of `id`. Thus, the first time `s` is evaluated `id` is bound to the value returned from the evaluation of the lower bound `e1` and so forth until the upper bound is reached ie. until `s` > `e2` . Note that `e1, e2` and `e3` are evaluated before entering the loop.

A *set for-loop statement* has the form:

```
for all e in set S do
s
```

where `S` is a set expression. The statement `s` is evaluated in the current environment extended with a binding of e to subsequent values from the set `S`.

A *sequence for-loop statement* has the form:

```
for e in l do
s
```

where l is a sequence expression. The statement s is evaluated in the current environment extended with a binding of e to subsequent values from the sequence l. If the keyword reverse is used the elements of the sequence l will be taken in reverse order.

Examples: The operation Remove demonstrates the use of a *sequence-for* loop to remove all occurences of a given number from a sequence of numbers:

```
Remove : (seq of nat) * nat ==> seq of nat
Remove (k,z) ==
(dcl nk : seq of nat := [];
 for elem in k do
   if elem <> z
   then nk := nk^[elem];
 return nk
);
```

A *set-for* loop can be exploited to return the set of winners of all groups:

```
GroupWinners: () ==> set of Team
GroupWinners () ==
(dcl winners : set of Team := {};
 for all gp in set dom gps do
   (dcl winner: Team := GroupWinner(gp);
    winners := winners union {winner}
   );
 return winners
 );
```

An example of a *index-for* loop is the classic bubblesort algorithm:

```
BubbleSort : seq of nat ==> seq of nat
BubbleSort (k) ==
  (dcl sorted_list : seq of nat := k;
   for i = len k to 1 by -1 do
     for j = 1 to i-1 do
       if sorted_list(j) > sorted_list(j+1)
       then (dcl temp:nat := sorted_list(j);
             sorted_list(j) := sorted_list(j+1);
             sorted_list(j+1) := temp
```

```
          );
  return sorted_list
  )
```

13.7 The While-Loop Statement

Syntax: statement = ...
| while loop
| ... ;

while loop = 'while', expression, 'do', statement ;

Semantics: The semantics for the *while statement* corresponds to the while statement from traditional programming languages. The form of a *while loop* is:

```
while e do
  s
```

where e is a boolean expression and s a statement. As long as the expression e evaluates to true the body statement s is evaluated.

Examples: The *while loop* can be illustrated by the following example which uses Newton's method to approximate the square root of a real number r within relative error e.

```
SquareRoot : real * real ==> real
SquareRoot (r,e) ==
  (dcl x:real := 1,
       nextx:real := r;
   while abs (x - nextx) >= e * x do
     ( x := nextx;
       nextx := ((r / x) + x) / 2;
     );
   return nextx
  );
```

13.8 The Nondeterministic Statement

Syntax: statement = ...
| nondeterministic statement
| ... ;

nondeterministic statement = ‘||’, ‘(’, statement,
{ ‘,’, statement }, ‘)’ ;

Semantics: The *nondeterministic statement* has the form:

```
|| (stmt1, stmt2, ..., stmtn)
```

and it represents the execution of the component statements `stmti` in an arbitrary (non-deterministic) order. However, it should be noted that the component statements are not executed simultaneously. Notice that the interpreter will use an underdetermined[21] semantics even though this construct is called a non-deterministic statement.

Examples: Using the state definition

```
state St of
  x:nat
  y:nat
  l:seq1 of nat
end
```

we can use the non-deterministic statement to effect a bubble sort:

```
Sort: () ==> ()
Sort () ==
  while x < y do
    ||(BubbleMin(), BubbleMax());
```

Here `BubbleMin` "bubbles" the minimum value in the subsequence `l(x,...,y)` to the head of the subsequence and `BubbleMax` "bubbles" the maximum value in the subsequence `l(x,...,y)` to the last index in the subsequence.

[21] Even though the user of the interpreter does not know the order in which these statements are executed they are always executed in the same order unless the seed option is used.

`BubbleMin` works by first iterating through the subsequence to find the index of the minimum value. The contents of this index are then swapped with the contents of the head of the list, `l(x)`.

```
BubbleMin : () ==> ()
BubbleMin () ==
  (dcl z:nat := x;
   dcl m:nat := l(z);
   -- find min val in l(x..y)
   for i = x to y do
     if l(i) < m
     then ( m := l(i);
            z := i);
   -- move min val to index x
   (dcl temp:nat;
    temp := l(x);
    l(x) := l(z);
    l(z) := temp;
    x := x+1));
```

`BubbleMax` operates in a similar fashion. It iterates through the subsequence to find the index of the maximum value, then swaps the contents of this index with the contents of the last element of the subsequence.

```
BubbleMax : () ==> ()
BubbleMax () ==
  (dcl z:nat := x;
   dcl m:nat := l(z);
   -- find max val in l(x..y)
   for i = x to y do
     if l(i) > m
     then ( m := l(i);
            z := i);
   -- move max val to index y
   (dcl temp:nat;
    temp := l(y);
    l(y) := l(z);
    l(z) := temp;
    y := y-1));
```

13.9 The Call Statement

Syntax: statement = ...
| call statement
| ... ;

call statement = name, '(', [expression list], ')' ;

Semantics: The *call statement* has the form:

```
opname(param1, param2, ..., paramn)
```

The *call statement* calls an operation, `opname`, and returns the result of evaluating the operation. Because operations can manipulate global variables a *call statement* does not necessarily have to return a value as function calls do.

Examples: The operation `ResetStack` given below does not have any parameter and does not return a value whereas the operation `PopStack` returns the top element of the stack.

```
ResetStack();
...
top := PopStack();
```

where `PopStack` could be defined as:

```
PopStack: () ==> Elem
PopStack() ==
  def res = hd stack in
   (stack := tl stack;
    return res)
pre stack <> []
post stack~ = [RESULT] ^ stack
```

where `stack` is a global variable.

13.10 The Return Statement

Syntax: statement = ...
| return statement
| ... ;

return statement = 'return', [expression] ;

Semantics: The *return statement* returns the value of an expression inside an operation. The value is evaluated in the given context. If an operation does not return a value, the expression must be omitted. A *return statement* has the form:

```
return e
```

or

```
return
```

where expression `e` is the return value of the operation.

Examples: In the following example `OpCall` is an operation call whereas `FunCall` is a function call. As the *if statement* only accepts statements in the two branches `FunCall` is "converted" to a statement by using the *return statement*.

```
if test
then  OpCall()
else  return FunCall()
```

13.11 Exception Handling Statements

Syntax: statement = ...
| always statement
| trap statement
| recursive trap statement
| exit statement
| ... ;

always statement = ‘always’, statement, ‘in’, statement ;

trap statement = ‘trap’, pattern bind, ‘with’, statement, ‘in’,
statement ;

recursive trap statement = ‘tixe’, traps, ‘in’, statement ;

traps = ‘{’, pattern bind, ‘|->’, statement,
{ ‘,’, pattern bind, ‘|->’, statement }, ‘}’ ;

exit statement = ‘exit’, [expression] ;

Semantics: The exception handling statements are used to control exception errors in a specification. This means that we have to be able to signal an exception within a specification. This can be done with the *exit statement*, and has the form:

```
exit e
```

or

```
exit
```

where `e` is an expression which is optional. The expression `e` can be used to signal what kind of exception is raised.

The *always statement* has the form:

```
always s1 in
s2
```

where `s1` and `s2` are statements. First statement `s2` is evaluated, and regardless of any exceptions raised, statement `s1` is also evaluated. The result value of the complete *always statement* is determined by the evaluation of statement `s1`: if this raises an exception, this value is returned, otherwise the result of the evaluation of statement `s2` is returned.

The *trap statement* only evaluates the handler statement, `s1`, when certain conditions are fulfilled. It has the form:

```
trap pat with s1 in s2
```

where `pat` is a pattern or bind used to select certain exceptions, `s1` and `s2` are statements. First, we evaluate statement `s2`, and if no exception is raised, the result value of the complete *trap statement* is the result of the evaluation of `s2`. If an exception is raised, the value of `s2` is matched against the pattern `pat`. If there is no matching, the exception is returned as result of the complete *trap statement*, otherwise, statement `s1` is evaluated and the result of this evaluation is also the result of the complete *trap statement*.

The *recursive trap statement* has the form:

```
tixe {
  pat1 |-> s1,
  ...
  patn |-> sn
} in s
```

where `pat1, ..., patn` are patterns or binds, `s, s1, ..., sn` are statements. First, statement `s` is evaluated, and if no exception is raised, the result is returned as the result of the complete *recursive trap statement*. Otherwise, the value is matched in order against each of the patterns `pati`. When a match cannot be found, the exception is returned as the result of the *recursive trap statement*. If a match is found, the corresponding statement `si` is evaluated. If this does not raise an exception, the result value of the evaluation of `si` is returned as the result of the *recursive trap statement*. Otherwise, the matching starts again, now with the new exception value (the result of the evaluation of `si`).

Examples: In many programs, we need to allocate memory for a single operation. After the operation is completed, the memory is not needed anymore. This can be done with the *always statement*:

```
( dcl mem : Memory;
  always Free(mem) in
  ( mem := Allocate();
    Command(mem, ...)
  )
)
```

In the above example, we cannot act upon a possible exception raised within the body statement of the *always statement*. By using the *trap statement* we can catch these exceptions:

```
trap pat with ErrorAction(pat) in
( dcl mem : Memory;
  always Free(mem) in
  ( mem := Allocate();
    Command(mem, ...)
  )
)
```

Now all exceptions raised within the *always statement* are captured by the *trap statement*. If we want to distinguish between several exception values, we can use either nested *trap statements* or the *recursive trap statement*:

```
DoCommand : () ==> int
DoCommand () ==
( dcl mem : Memory;
  always Free(mem) in
  ( mem := Allocate();
    Command(mem, ...)
  )
);

Example : () ==> int
Example () ==
tixe
{ <NOMEM> |-> return -1,
  <BUSY>  |-> DoCommand(),
  err     |-> return -2 }
in
  DoCommand()
```

In operation `DoCommand` we use the *always statement* in the allocation of memory, and all exceptions raised are captured by the *recursive trap statement* in operation `Example`. An exception with value `<NOMEM>` results in a return value of `-1` and no exception raised. If the value of the exception is `<BUSY>` we try to perform the operation `DoCommand` again. If this raises an exception, this is also handled by the *recursive trap statement*. All other exceptions result in the return of the value `-2`.

13.12 The Error Statement

Syntax: statement = ...
| error statement
| ... ;

error statement = 'error' ;

Semantics: The *error statement* corresponds to the undefined expression. It is used to state explicitly that the result of a statement is undefined and because of this an error has occurred. When an *error statement* is evaluated the interpreter will terminate the execution of the specification and report that an *error statement* was evaluated.

Pragmatically use of error statements differs from pre-conditions as was the case with undefined expressions: use of a pre-condition means it is the caller's responsibility to ensure that the pre-condition is satisfied when the operation is called; if an error statement is used it is the called operation's responsibility to deal with error handling.

Examples: The operation `SquareRoot` on page 91 does not exclude the possibility that the number to be square rooted might be negative. We remedy this in the operation `SquareRootErr`:

```
SquareRootErr : real * real ==> real
SquareRootErr (r,e) ==
  if r < 0
  then error
  else
    (dcl x:real := 1;
     dcl nextx:real := r;
     while abs (x - nextx) >= e * x do
       ( x := nextx;
         nextx := ((r / x) + x) / 2;
       );
     return nextx
    )
```

13.13 The Identity Statement

Syntax: statement = ...
| identity statement ;

identity statement = ‘skip’ ;

Semantics: The *identity statement* is used to signal that no evaluation takes place.

Examples: In the operation Remove in section 13.6 the behaviour of the operation within the for loop if elem=z is not explicitly stated. Remove2 below does this.

```
Remove2 : (seq of nat) * nat ==> seq of nat
Remove2 (k,z) ==
  (dcl nk : seq of nat := [];
   for elem in k do
     if elem <> z then nk := nk^[elem]
     else skip;
   return nk
  );
```

Here, we explicitly included the else-branch to illustrate the *identity statement*, however, in most cases the else-branch will not be included and the *identity statement* is implicitly assumed.

13.14 The Specification Statement

Syntax: statement = ...
| specification statement ;

specification statement = ‘[’, implicit operation body, ‘]’ ;

Semantics: The specification statement can be used to describe a desired effect a statement in terms of a pre- and a post-condition. Thus, it captures the abstraction of a statement, permitting it to have an abstract (implicit) specification without being forced to an operation definition. The specification statement is equivalent with the body of an implicitly defined operation (see section 12). Thus specification statements can not be executed.

Examples: We can use a specification statement to specify a bubble maximum part of a bubble sort:

```
Sort2 : () ==> ()
Sort2 () ==
  while x < y do
    || (BubbleMin(),
        [ext wr l : seq1 of nat
             wr y : nat
             rd x : nat
         pre x < y
         post y < y~ and
                permutation (l~(x,...,y~),l(x,...,y~)) and
                forall i in set {x,...,y} & l(i) < l(y~)]
       )
```

(permutation is an auxiliary function taking two sequences which returns true iff one sequence is a permutation of the other.)

14 Top-level Specification

In the previous sections all the VDM-SL constructs such as types, expressions, statements, functions and operations have been described. A number of these constructs can constitute a top-level VDM-SL specification. A top-level specification can be created in two ways:

1. The specification is split into a number of modules which are specified separately, but can depend on each other.
2. The specification is specified in a flat manner, i.e. no modules are used.

Thus, a complete specification, or document, has the following syntax.

Syntax: document = any module, { any module }
| definition block, { definition block } ;

any module = module
| dynamic link module ;

14.1 A Flat Specification

As said, a flat specification does not use modules. This means that all constructs can be used throughout the specification. In the flat case, a document has a syntax of:

document = ...
| definition block, { definition block } ;

definition block = type definitions
| state definition
| value definitions
| function definitions
| operation definitions ;

Thus, a flat specification is made up of several *definition* blocks. However, only one state definition is allowed. The following is an example of a flat top-level specification:

```
values

  st1 = mk_St([3,2,-9,11,5,3])

state St of
  l:seq1 of nat
end

functions

min_index : seq1 of nat -> nat
min_index(l) ==
  if len l = 1
  then 1
  else let mi = min_index(tl l)
      in if l(mi+1) < hd l
        then mi+1
        else 1

operations

SelectionSort : nat ==> ()
SelectionSort (i) ==
  if i < len l
  then (dcl temp: nat;
        dcl mi : nat := min_index(l(i,...,len l)) + i - 1;

        temp := l(mi);
        l(mi) := l(i);
        l(i) := temp;
        SelectionSort(i+1)
       )
```

14.2 A Structured Specification

As an extension to the standard VDM-SL language, it is possible to structure an VDM-SL specification using modules. In this section, the use of modules to create the top-level specification will be described. With the structuring facilities offered by VDM-SL it is possible to:

- Export constructs from a module.
- Import constructs from a module.
- Rename constructs upon import.
- Define a state in a module.

In addition to these kinds of ordinary modules it is possible to use so-called "Dynamic Link Modules" (see section 15).

14.2.1 The Layout of a Module

Before the actual facilities are described, the general layout of a module is described. A module consists of three parts: a *module declaration*, an *interface section*, and a *definitions section*. It is possible to leave out the definitions part in the early development of a module specification.

In the module declaration, the module is named. The name must be a unique module name within the complete specification. The second part, the interface section, defines the relation of a module with other modules and consists of a number of sections. These sections are:

- An *imports section*. In the imports section, all the constructs that are going to be used from other modules are described. If constructs are going to be renamed it has to be done in the imports section.
- An *exports section*. Here all the constructs that are going to be used in other modules are defined. If no exports section is present the module cannot be used from other modules.

The third part of a module declaration, the definitions section, contains all the definitions of the module. Thus, in general, the syntax of a module is:

Syntax: module = 'module', identifier, interface,
[module body], 'end', identifier ;

module body = 'definitions', definition block, { definition block } ;

To illustrate the use of modules, the example flat top-level specification are rewritten with some minor modifications. Some unimportant parts of the flat specification are left out for clarity.

14.2.2 The Exports Section

Syntax: interface = [import definition list],
export definition ;

export definition = 'exports', export module signature ;

export module signature = 'all'
| export signature,
{ export signature } ;

export signature = export types signature
| values signature
| export functions signature
| operations signature ;

export types signature = 'types', type export,
{ ';', type export }, [';'] ;

type export = ['struct'], name ;

values signature = 'values', value signature,
{ ';', value signature }, [';'] ;

value signature = name list, ':', type ;

export functions signature = 'functions' function export,
{ ';', function export } ;

function export = name list, [type variable list], ':',
function type ;

functions signature = 'functions' function signature,
{ ';', function signature }, [';'] ;

function signature = name list, ':', function type ;

operations signature = 'operations' operation signature,
{ ';', operation signature }, [';'] ;

operation signature = name list, ':', operation type ;

Semantics: The exports section must be used to make constructs visible to other modules. Some or all of the defined constructs from a module can be exported. In the latter case, the keyword all is used. However, imported constructs are not exported from the module. If only part of the constructs are exported, the visible constructs with the appropriate signatures are stated.

Normally, if a construct is visible to another module, that construct can be considered to be defined inside the module. However, with types and operations there are some exceptions:

Types: If a type `T` is defined in module `A` and this type is also going to be used in module `B`, the type from module `A` has to be exported. This can be done in two ways:

1. The name of the type is exported.
2. The structure of the type is exported.

If only the name of the type is exported, the other module cannot create values of type `T`. This means that the exporting module (`A`) must provide functions and/or operations to directly create and manipulate values of type `T` by means of the constructors related to the representation of `T`.

If we export the structure of the type by using the keyword struct, the other module can create and manipulate values of type `T` (it can also use mk_ keyword and the is_ keyword for this type if it is a record type).

If the type also defines an invariant, the invariant predicate function is only exported if the structure of the type is exported.

Operations: In a module, a state that is global for the module can be defined. All operations within the module can manipulate that state. If operations are exported from a module, they manipulate the state in the exporting module, i.e. the state in the module where they are defined.

If an exported function or an operation defines a pre- and/or post-condition, the corresponding predicate functions (see section 6) are also exported.

Examples: Consider a model of a bank account. An account is characterised by the name of the holder, the account number, the bank branch at which the account is maintained, the balance, and an encrypted PIN code for the ATM card. We might model this as follows:

```
module BankAccount
```

```
exports types digit; account
        functions digval: digit -> nat;
                withdrawal: account * real -> account;
                isPin: account * nat -> bool;
                requestWithdrawal: account * nat -> bool
definitions

types

digit = nat
inv d == d < 10;

account::  holder : seq1 of char
           number : seq1 of digit
           branchcode : seq1 of digit
           balance: real
           epin: nat
inv mk_account(holder, number, branchcode,-,-) ==
  len number = 8 and len branchcode = 6

functions

  digval : digit -> nat
  digval(d) == d;

  deposit: account * real -> account
  deposit(acc,r) ==
    mu(acc,balance |-> acc.balance + r);

  withdrawal : account * real -> account
  withdrawal (acc,r) ==
    mu(acc,balance |-> acc.balance - r);

  isPin : account * nat -> bool
  isPin(acc,ep) ==
    ep = acc.epin;

  requestWithdrawal : account * nat -> bool
  requestWithdrawal (acc,amt) ==
    acc.balance > amt
```

```
end BankAccount
```

In this module we export two types and five functions. Note that since we have enumerated the entities we are exporting, but have not exported `digit` or `account` using the struct keyword, the internals of `account` values may not be accessed by other modules, neither may the invariant for `digit`. If such access is necessary, the types should be exported with the struct keyword, or all constructs in the module should be exported using the exports all clause.

The module `Keypad` given below models the keypad interface of an ATM machine. The state variable maintains a buffer of data typed at the keypad by the user.

```
module Keypad

  imports
    from BankAccount types digit

  exports all

  definitions

  state buffer of
    data : seq of BankAccount`digit
  end

  operations

    DataAvailable : () ==> bool
    DataAvailable () ==
      return(data <> []);

    ReadData : () ==> seq of BankAccount`digit
    ReadData () ==
      return(data);

    WriteData : seq of BankAccount`digit ==> ()
    WriteData (d) ==
      data := data^d
```

```
end Keypad
```

In this module all constructs are exported. Since the only entities defined are the state and operations on it, this means that all of the operations may be accessed by an importing module. The state is not accessible to importing modules, but remains private to this module. However the state constructor `mk_Keypad‘buffer` is accessible.

14.2.3 The Imports Section

Syntax: interface = [import definition list],
export definition ;

import definition list = ‘imports’, import definition,
{ ‘,’, import definition } ;

import definition = ‘from’, identifier, import module signature ;

import module signature = ‘all’
| import signature,
{ import signature } ;

import signature = import types signature
| import values signature
| import functions signature
| import operations signature ;

import types signature = ‘types’, type import,
{ ‘;’, type import }, [‘;’] ;

type import = name, [‘renamed’, name]
| type definition, [‘renamed’, name] ;

import values signature = ‘values’, value import,
{ ‘;’, value import }, [‘;’] ;

value import = name, [‘:’, type], [‘renamed’, name] ;

import functions signature = ‘functions’, function import,
{ ‘;’, function import }, [‘;’] ;

function import = name, [[type variable list],
‘:’, function type], [‘renamed’, name] ;

import operations signature = ‘operations’, operation import,
{ ‘;’, operation import }, [‘;’] ;

operation import = name, [‘:’, operation type],
[‘renamed’, name] ;

Semantics: The imports section is used to state what constructs are used from other modules with the restriction that only visible constructs can be imported. If all the visible constructs from a module are going to be used, the keyword all is used, unless one or more constructs are going to be renamed. With renaming, an imported construct is given a new name which can be used instead of the original name preceded by the exporting module name. In general this has the form:

```
name renamed new_name
```

where `name` is the name of the imported construct, and `new_name` is the new name for the construct. This way, more meaningful names can be given to constructs. Note that in the importing module it is not possible to refer to `DefModule‘name` (where `DefModule` is the name of the defining module) any longer but only to `newname`.

It is possible to include type information in the imports section, such that this information will only be used by the static semantics check of the complete module. If no type information is given, the static semantics can also find this information in the exporting module (see section 17).

When a type which has been exported with the struct keyword (with its structure) is imported the importing module may only make use of this structure if it repeats the type definition from the exporting module in its type import. In case such a type is a composite type and it is also renamed this has the consequence that the tag is renamed as well.

Examples: We can model an ATM card as consisting of a card number and an expiry date. This requires the `digit` type defined in the module `BankAccount`. It also uses the function `digval` from the same module.

```
module ATMCard
```

```
imports
  from BankAccount types digit
                   functions digval renamed atmc_digval

exports all

definitions

types

  digit = BankAccount`digit;

  atmc:: cardnumber : seq1 of digit
         expiry : digit * digit * digit * digit
  inv mk_atmc(cardnumber, mk_(m1,m2,-,-)) ==
      atmc_digval(m1) * 10 + atmc_digval(m2) <= 12 and
      len cardnumber >= 8

functions

  getCardnumber : atmc -> seq1 of digit
  getCardnumber (atmc) ==
    atmc.cardnumber

end ATMCard
```

Here the invariant on the type `atmc` states that expiry dates must represent valid dates, and card numbers must be at least 8 digits long. Note that since `digit` is not exported with the struct keyword from the module `BankAccount`, we cannot access the invariant for `digit` in module `ATMCard`. However this notwithstanding, all values of type `digit` manipulated in `ATMCard` must satisfy the invariant.

15 Dynamic Link Modules

Dynamic Link modules are used to describe the interface between modules which are fully specified in VDM-SL and parts of the overall system which are only available as C++ code. This facility enables users to make use of existing C++ libraries while a specification is being interpreted/debugged. The usage of this facility is described in detail in [?]. The general layout of a Dynamic Link module is similar to an ordinary VDM-SL module. It has three parts: a *module declaration*, an *interface section*, and an optional *library reference*.

Syntax: The module declaration of a Dynamic Link module is simply the keywords dlmodule followed by the name of the module. The interface section of a Dynamic Link module is simpler than the interface section for an ordinary module. The only kind of constructs which can be imported into a Dynamic Link module are types. Such imported types can be used in the signature of the values, functions and/or operations which are exported from the module. Finally the library reference (identified by the 'uselib' keyword) is used to identify the dynamically linked C++ library which must be used by the interpreter in case a specification which makes use of code from such a library is going to be interpreted.

The syntax for Dynamic Link modules is:

dynamic link module = 'dlmodule', identifier,
dynamic link interface,
[use signature],
'end', identifier ;

dynamic link interface = [dynamic link import definition list],
dynamic link export definition ;

use signature = 'uselib', text literal ;

dynamic link import definition list = 'imports',
dynamic link import definition,
{ ',', dynamic link import definition } ;

dynamic link import definition = 'from', identifier,
dynamic link import types signatures ;

dynamic link import types signatures = 'types', name,
{ ';', name }, [';'] ;

dynamic link export definition = 'exports',
dynamic link export signature,
{ dynamic link export signature } ;

dynamic link export signature = values signature
| functions signature
| operations signature ;

Semantics: The semantics of the interface constructs is identical to the semantics of these parts for ordinary modules. The semantics of the use signature is given by the C++ compiler which has been used to create the dynamically linked C++ libraries. Thus, the C++ code referred to in the use signature is not provided with semantics directly at the VDM-SL level.

Example: The example presented here is used in [?]. The first module imports constructs from a `MATH` module and a `CYLIO` module. Both of these other modules are presented afterwards and both of them are Dynamic Link modules.

```
module CYLINDER
  imports
    from MATH
        functions
          ExtSin : real -> real
        values
          ExtPI : real,

    from CYLIO
        functions
          ExtGetCylinder : () -> CircCyl

        operations
          ExtShowCircCylVol : CircCyl * real ==> ()

   exports
     types
         CircCyl

  definitions
    types
        CircCyl :: rad    : real
                   height : real
                   slope  : real

    functions
        CircCylVol : CircCyl -> real
        CircCylVol(cyl) ==
          MATH`ExtPI * cyl.rad * cyl.rad * cyl.height *
          MATH`ExtSin(cyl.slope)
```

```
    operations
        CircCyl : () ==> ()
        CircCyl() == ( let cyl = CYLIO`ExtGetCylinder() in
                              let vol = CircCylVol(cyl) in
                                 CYLIO`ExtShowCircCylVol(cyl, vol))
end CYLINDER
```

The MATH module is defined as:

```
dlmodule MATH
  exports
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

   values
      ExtPI : real

   uselib
      "libmath.so"

end MATH
```

The CYLIO module is defined as:

```
dlmodule CYLIO
  imports
    from CYLINDER
      types
        CircCyl

   exports
     functions
       ExtGetCylinder : () -> CircCyl

     operations
       ExtShowCircCylVol : CircCyl * real ==> ()

   uselib
      "libcylio.so"

end CYLIO
```

The way to use such modules with the VDM-SL Interpreter is described in [?]

16 Differences between VDM-SL and ISO/VDM-SL

This version of VDM-SL is based on the ISO/VDM-SL standard, but a few differences exist. These differences are both syntactical and semantical, and are mainly due to the extensions of the language and to requirements to make VDM-SL constructs executable[22].

The major difference between VDM-SL and ISO/VDM-SL is the availability of a structuring in VDM-SL. This causes some syntactical differences.

For the flat part of VDM-SL, the following differences with ISO/VDM-SL exist:

Syntactical differences:

- Semicolon (";") is used in the standard as a separator between subsequent constructs (e.g., between function definitions). VDM-SL adds to this rule that an optional semicolon can be put after the last of such a sequence of constructs. This change apply to the following syntactic definitions (see appendix A): *state definition*, *type definitions*, *values definitions*, *function definitions*, *operation definitions*, *def expression*, *def statement*, and *block statement*.
- In explicit function and operation definitions it is possible to specify an optional post condition in VDM-SL (see section 6 and section 12 or section A.3.4 or section A.3.5).
- The body of explicit function and operation definitions can be specified in a preliminary manner using the clause `is not yet specified`.
- An extended form for explicit function and operation definitions has been included. The extension is to enable the function and operation definition to use a heading similar to that used for implicit definitions. This makes it easier first to write an implicit definition and then add an algorithmic part later on. In addition the result identifier type pair has been generalised to work with more than one identifier.

[22] The semantics mentioned here is the semantics of the interpreter.

- In a flat specification the keyword definitions is not used. This way, a flat specification can be distributed over several files. However, in a module, the definitions section must begin with the keyword definitions (see Section 14.1).
- VDM-SL has been extended with the *specification statement*.
- In an *if statement* the "else" part is optional (see section 13.5 or section A.7.3).
- An empty set and an empty sequence can be used directly as patterns (see section 8 or section A.8.1).
- "map domain restrict to" and "map domain restrict by" have a right grouping (see section C.7).
- The operator precedence ordering for map type constructors is different from the standard (see section C.8).

Semantical differences (wrt. the interpreter):

- VDM-SL only operates with a conditional logic (see section 4.1.1).
- The initialisation of a global state must be written in a special constructive way. Note that the state of a module is only initialised if at least one operation from that module is used (see section 11).
- In VDM-SL , *value definitions* which are mutually recursive cannot be executed and they must be ordered such that they are defined before they are used (see section 10).
- The local definitions in a *let statement* and a *let expression* cannot be recursively defined. Furthermore they must be ordered such that they are defined before they are used (see section 7.1 and section 13.1).
- The numeric type rat in VDM-SL denotes the same type as the type real (see section 4.1.2).
- The two forms of interpreting looseness which are used in ISO/VDM-SL are 'underdeterminedness' and 'nondeterminism'. In ISO/VDM-SL the looseness in operations is nondeterministic whereas it is underdetermined for functions. In VDM-SL the looseness in both operations and functions is underdetermined. This is, however, also in line with the standard because the interpreter simply corresponds to one of the possible models for a specification.

17 Static Semantics

VDM specifications that are syntactically correct according to the syntax rules do not necessarily obey the typing and scoping rules of the language. The well-formedness of a VDM specification can be checked by the *static semantics checker*. In the Toolbox such a static semantics checker (for programming languages this is normally referred to as a type checker) is also present.

In general, it is not statically decidable whether a given VDM specification is well-formed or not. The static semantics for VDM-SL differs from the static semantics of other languages in the sense that it only rejects specifications which are definitely not well-formed, and only accepts specifications which are definitely well-formed. Thus, the static semantics for VDM-SL attaches a *well-formedness grade* to a VDM specification. Such a well-formedness grade indicates whether a specification is definitely well-formed, definitely not-well-formed, or possibly well-formed.

In the Toolbox this means that the static semantics checker can be called for either possible correctness or definite correctness. However, it should be noted that only very simple specifications will be able to pass the definite well-formedness check. Thus, for practical use the possible well-formedness is most useful.

The difference between a possibly well-formedness check and a definite well-formedness check can be illustrated by the following fragment of a VDM specification:

```
if a = true
then a + 1
else not a
```

where a has the type nat | bool (the union type of nat and bool). The reader can easily see that this expression is ill-formed if a is equal to true because then it will be impossible to add one to a. However, since such expressions can be arbitrarily complex this can in general not be checked statically. In this particular example possible well-formedness will yield true while definite well-formedness will yield false.

References

[1] Bicarregui, J., Fitzgerald, J., Lindsay, P., Moore, R., and Ritchie, B. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[2] Dawes, J. *The VDM-SL Reference Guide*. Pitman, 1991. ISBN 0-273-03151-1.

[3] Fitzgerald, J., and Jones, C. *Proof in VDM: case studies*. Springer-Verlag FACIT Series, 1998, ch. Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant.

[4] Jones, C. B. *Systematic Software Development Using VDM*, second ed. Prentice-Hall International, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-880733-7.

[5] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[6] Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1991.

A The VDM-SL Syntax

This appendix specifies the complete syntax for VDM-SL.

A.1 Document

document = any module, { any module }
| definition block, { definition block } ;

any module = module
| dynamic link module ;

A.2 Modules

This entire subsection is not present in the current version of the VDM-SL standard. Non standard

module = 'module', identifier, interface,
[module body], 'end', identifier ;

interface = [import definition list],
export definition ;

import definition list = 'imports', import definition,
{ ',', import definition } ;

import definition = 'from', identifier, import module signature ;

import module signature = 'all'
| import signature, { import signature } ;

import signature = import types signature
| import values signature
| import functions signature
| import operations signature ;

import types signature = 'types', type import,
{ ';', type import }, [';'] ;

type import = name, ['renamed', name]
| type definition, ['renamed', name] ;

import values signature = 'values', value import,
{ ';', value import }, [';'] ;

value import = name, [':', type], ['renamed', name] ;

import functions signature = 'functions', function import,
{ ';', function import }, [';'] ;

function import = name, [[type variable list], ':', function type],
['renamed', name] ;

import operations signature = 'operations', operation import,
{ ';', operation import }, [';'] ;

operation import = name, [':', operation type], ['renamed', name] ;

export definition = 'exports', export module signature ;

export module signature = 'all'
| export signature,
{ export signature } ;

export signature = export types signature
| values signature
| export functions signature
| operations signature ;

export types signature = 'types', type export,
{ ';', type export }, [';'] ;

type export = [‘struct’], name ;

values signature = ‘values’, value signature,
{ ‘;’, value signature }, [‘;’] ;

value signature = name list, ‘:’, type ;

export functions signature = ‘functions’ function export,
{ ‘;’, function export } ;

function export = name list, [type variable list], ‘:’,
function type ;

functions signature = ‘functions’ function signature,
{ ‘;’, function signature }, [‘;’] ;

function signature = name list, ‘:’, function type ;

operations signature = ‘operations’ operation signature,
{ ‘;’, operation signature }, [‘;’] ;

operation signature = name list, ‘:’, operation type ;

dynamic link module = ‘dlmodule’, identifier,
dynamic link interface,
[use signature],
‘end’, identifier ;

dynamic link interface = [dynamic link import definition list],
dynamic link export definition ;

use signature = ‘uselib’, text literal ;

dynamic link import definition list = ‘imports’,
dynamic link import definition,
{ ‘,’, dynamic link import definition } ;

dynamic link import definition = ‘from’, identifier,
dynamic link import types signatures ;

dynamic link import types signatures = ‘types’, name,
{ ‘;’, name }, [‘;’] ;

dynamic link export definition = ‘exports’,
dynamic link export signature,
{ dynamic link export signature } ;

dynamic link export signature = values signature
| functions signature
| operations signature ;

A.3 Definitions

module body = ‘definitions’, definition block, { definition block } ;

definition block = type definitions
| state definition
| value definitions
| function definitions
| operation definitions ;

A.3.1 Type Definitions

type definitions = ‘types’, [type definition ,
{ ‘;’, type definition }, [‘;’]] ; Non standard

type definition = identifier, ‘=’, type, [invariant]
| identifier, ‘::’, field list, [invariant] ;

type = bracketed type
| basic type
| quote type

| composite type
| union type
| product type
| optional type
| set type
| seq type
| map type
| partial function type
| type name
| type variable ;

bracketed type = ‘(’, type, ‘)’ ;

basic type = ‘bool’ | ‘nat’ | ‘nat1’ | ‘int’ | ‘rat’
| ‘real’ | ‘char’ | ‘token’ ;

quote type = quote literal ;

composite type = ‘compose’, identifier, ‘of’, field list, ‘end’ ;

field list = { field } ;

field = [identifier, ‘:’], type
| [identifier, ‘:-’], type ;

union type = type, ‘|’, type, { ‘|’, type } ;

product type = type, ‘*’, type, { ‘*’, type } ;

optional type = ‘[’, type, ‘]’ ;

set type = ‘set of’, type ;

seq type = seq0 type
| seq1 type ;

seq0 type = ‘seq of’, type ;

seq1 type = ‘seq1 of’, type ;

map type = general map type
| injective map type ;

general map type = ‘map’, type, ‘to’, type ;

injective map type = ‘inmap’, type, ‘to’, type ;

function type = partial function type
| total function type ;

partial function type = discretionary type, ‘->’, type ;

total function type = discretionary type, ‘+>’, type ;

discretionary type = type
| ‘(’, ‘)’ ;

type name = name ;

type variable = type variable identifier ;

A.3.2 The State Definition

state definition = ‘state’, identifier, ‘of’, field list,
[invariant], [initialisation], ‘end’, [‘;’] ; Non standard

invariant = ‘inv’, invariant initial function ;

initialisation = ‘init’, invariant initial function ;

invariant initial function = pattern, ‘==’, expression ;

A.3.3 Value Definitions

value definitions = ‘values’, [value definition,
{ ‘;’, value definition }, [‘;’]] ; Non standard

value definition = pattern, [‘:’, type], ‘=’, expression ; Non standard

A.3.4 Function Definitions

function definitions = ‘functions’, [function definition,
{ ‘;’, function definition }, [‘;’]] ; Non standard

function definition = explicit function definition
| implicit function definition
| extended explicit function definition ; Non standard

explicit function definition = identifier, [type variable list], ‘:’,
function type,
identifier, parameters list,
‘==’, function body,
[‘pre’, expression] ,
[‘post’, expression],
[‘measure’, name] ; Non standard

implicit function definition = identifier, [type variable list],
parameter types,
identifier type pair list,
[‘pre’, expression],
‘post’, expression ;

extended explicit function definition = identifier, [type variable list], Non standard
parameter types,
identifier type pair list,
‘==’, function body,
[‘pre’, expression],
[‘post’, expression] ;

type variable list = ‘[’, type variable identifier,
{ ‘,’, type variable identifier }, ‘]’ ;

identifier type pair = identifier, ‘:’, type ;

parameter types = ‘(’, [pattern type pair list], ‘)’ ;

identifier type pair list = identifier, ‘:’, type,
{ ‘,’, identifier, ‘:’, type } ;

pattern type pair list = pattern list, ‘:’, type,
{ ‘,’, pattern list,‘:’, type } ;

parameters list = parameters, { parameters } ;

parameters = ‘(’, [pattern list], ‘)’ ;

function body = expression
| ‘is not yet specified’ ; Non standard

A.3.5 Operation Definitions

operation definitions = ‘operations’, [operation definition,
{ ‘;’, operation definition }, [‘;’]] ; Non standard

operation definition = explicit operation definition
| implicit operation definition
| extended explicit operation definition ; Non standard

explicit operation definition = identifier, ‘:’, operation type,
identifier, parameters,
‘==’, operation body,
[‘pre’, expression],
[‘post’, expression] ;

Non standard

implicit operation definition = identifier, parameter types,
[identifier type pair list],
implicit operation body ;

implicit operation body = [externals],
['pre', expression],
'post', expression,
[exceptions] ;

extended explicit operation definition = identifier, parameter types, Non standard
[identifier type pair list],
'==', operation body,
[externals],
['pre', expression],
['post', expression],
[exceptions] ;

operation type = discretionary type, '==>', discretionary type ;

operation body = statement
| 'is not yet specified' ; Non standard

externals = 'ext', var information, { var information } ;

var information = mode, name list, [':', type] ;

mode = 'rd' | 'wr' ;

exceptions = 'errs', error list ;

error list = error, { error } ;

error = identifier, ':', expression, '->', expression ;

A.4 Expressions

expression list = expression, { ‘,’, expression } ;

expression = bracketed expression
| let expression
| let be expression
| def expression
| if expression
| cases expression
| unary expression
| binary expression
| quantified expression
| iota expression
| set enumeration
| set comprehension
| set range expression
| sequence enumeration
| sequence comprehension
| subsequence
| map enumeration
| map comprehension
| tuple constructor
| record constructor
| record modifier
| apply
| field select
| tuple select
| function type instantiation
| lambda expression
| general is expression
| undefined expression
| precondition expression   Non standard
| name
| old name
| symbolic literal ;

A.4.1 Bracketed Expressions

bracketed expression = ‘(’, expression, ‘)’ ;

A.4.2 Local Binding Expressions

let expression = ‘let’, local definition, { ‘,’, local definition },
‘in’, expression ;

let be expression = ‘let’, bind, [‘be’, ‘st’, expression], ‘in’,
expression ;

def expression = ‘def’, pattern bind, ‘=’, expression,
{ ‘;’, pattern bind, ‘=’, expression }, [‘;’], Non standard
‘in’, expression ;

A.4.3 Conditional Expressions

if expression = ‘if’, expression, ‘then’, expression,
{ elseif expression },
‘else’, expression ;

elseif expression = ‘elseif’, expression, ‘then’, expression ;

cases expression = ‘cases’, expression, ‘:’,
cases expression alternatives,
[‘,’, others expression], ‘end’ ;

cases expression alternatives = cases expression alternative,
{ ‘,’, cases expression alternative } ;

cases expression alternative = pattern list, ‘->’, expression ;

others expression = ‘others’, ‘->’, expression ;

A.4.4 Unary Expressions

unary expression = prefix expression
| map inverse ;

prefix expression = unary operator, expression ;

unary operator = unary plus
| unary minus
| arithmetic abs
| floor
| not
| set cardinality
| finite power set
| distributed set union
| distributed set intersection
| sequence head
| sequence tail
| sequence length
| sequence elements
| sequence indices
| distributed sequence concatenation
| map domain
| map range
| distributed map merge ;

unary plus = ‘+’ ;

unary minus = ‘−’ ;

arithmetic abs = ‘abs’ ;

floor = ‘floor’ ;

not = ‘not’ ;

set cardinality = ‘card’ ;

finite power set = ‘power’ ;

distributed set union = ‘dunion’ ;

distributed set intersection = ‘dinter’ ;

sequence head = ‘hd’ ;

sequence tail = ‘tl’ ;

sequence length = ‘len’ ;

sequence elements = ‘elems’ ;

sequence indices = ‘inds’ ;

distributed sequence concatenation = ‘conc’ ;

map domain = ‘dom’ ;

map range = ‘rng’ ;

distributed map merge = ‘merge’ ;

map inverse = ‘inverse’, expression ;

A.4.5 Binary Expressions

binary expression = expression, binary operator, expression ;

binary operator = arithmetic plus
| arithmetic minus
| arithmetic multiplication
| arithmetic divide
| arithmetic integer division
| arithmetic rem
| arithmetic mod
| less than
| less than or equal

| greater than
| greater than or equal
| equal
| not equal
| or
| and
| imply
| logical equivalence
| in set
| not in set
| subset
| proper subset
| set union
| set difference
| set intersection
| sequence concatenate
| map or sequence modify
| map merge
| map domain restrict to
| map domain restrict by
| map range restrict to
| map range restrict by
| composition
| iterate ;

arithmetic plus = '+' ;

arithmetic minus = '-' ;

arithmetic multiplication = '*' ;

arithmetic divide = '/' ;

arithmetic integer division = 'div' ;

arithmetic rem = 'rem' ;

arithmetic mod = 'mod' ;

less than = ‘<’ ;

less than or equal = ‘<=’ ;

greater than = ‘>’ ;

greater than or equal = ‘>=’ ;

equal = ‘=’ ;

not equal = ‘<>’ ;

or = ‘or’ ;

and = ‘and’ ;

imply = ‘=>’ ;

logical equivalence = ‘<=>’ ;

in set = ‘in set’ ;

not in set = ‘not in set’ ;

subset = ‘subset’ ;

proper subset = ‘psubset’ ;

set union = ‘union’ ;

set difference = ‘\’ ;

set intersection = ‘inter’ ;

sequence concatenate = ‘^’ ;

map or sequence modify = ‘++’ ;

map merge = ‘munion’ ;

map domain restrict to = ‘<:’ ;

map domain restrict by = ‘<-:’ ;

map range restrict to = ‘:>’ ;

map range restrict by = ‘:->’ ;

composition = ‘comp’ ;

iterate = ‘**’ ;

A.4.6 Quantified Expressions

quantified expression = all expression
| exists expression
| exists unique expression ;

all expression = ‘forall’, bind list, ‘&’, expression ;

exists expression = ‘exists’, bind list, ‘&’, expression ;

exists unique expression = ‘exists1’, bind, ‘&’, expression ;

A.4.7 The Iota Expression

iota expression = ‘iota’, bind, ‘&’, expression ;

A.4.8 Set Expressions

set enumeration = ‘{’, [expression list], ‘}’ ;

set comprehension = ‘{’, expression, ‘|’, bind list,
[‘&’, expression], ‘}’ ;

set range expression = ‘{’, expression, ‘,’, ‘...’, ‘,’,
expression, ‘}’ ;

A.4.9 Sequence Expressions

sequence enumeration = ‘[’, [expression list], ‘]’ ;

sequence comprehension = ‘[’, expression, ‘|’, set bind,
[‘&’, expression], ‘]’ ;

subsequence = expression, ‘(’, expression, ‘,’, ‘...’, ‘,’,
expression, ‘)’ ;

A.4.10 Map Expressions

map enumeration = ‘{’, maplet, { ‘,’, maplet }, ‘}’
| ‘{’, ‘|->’, ‘}’ ;

maplet = expression, ‘|->’, expression ;

map comprehension = ‘{’, maplet, ‘|’, bind list,
[‘&’, expression], ‘}’ ;

A.4.11 The Tuple Constructor Expression

tuple constructor = ‘mk_’, ‘(’, expression, ‘,’, expression list, ‘)’ ;

A.4.12 Record Expressions

record constructor = ‘mk_’,[23] name, ‘(’, [expression list], ‘)’ ;

record modifier = ‘mu’, ‘(’, expression, ‘,’,
record modification,
{ ‘,’, record modification }, ‘)’ ;

record modification = identifier, ‘|->’, expression ;

A.4.13 Apply Expressions

apply = expression, ‘(’, [expression list], ‘)’ ;

field select = expression, ‘.’, identifier ;

tuple select = expression, ‘.#’, numeral ;

function type instantiation = name, ‘[’, type, { ‘,’, type }, ‘]’ ;

A.4.14 The Lambda Expression

lambda expression = ‘lambda’, type bind list, ‘&’, expression ;

A.5 The narrow Expression

narrow expression = ‘narrow_’, ‘(’, expression, ‘,’, type, ‘)’ ;

A.5.1 The Is Expression

general is expression = is expression
| type judgement ;

[23] **Note:** no delimiter is allowed

is expression = ‘is_’,[24] name, ‘(’, expression, ‘)’
| is basic type, ‘(’, expression, ‘)’ ;

type judgement = ‘is_’, ‘(’, expression, ‘,’, type, ‘)’ ;

A.5.2 The Undefined Expression

undefined expression = ‘undefined’ ; Non standard

A.5.3 The Precondition Expression

pre-condition expression = ‘pre_’, ‘(’, expression,
[{ ‘,’, expression }], ‘)’ ;

A.5.4 Names

name = identifier, [‘‘’, identifier] ;

name list = name, { ‘,’, name } ;

old name = identifier, ‘~’ ;

A.6 State Designators

state designator = name
| field reference
| map or sequence reference ;

field reference = state designator, ‘.’, identifier ;

map or sequence reference = state designator, ‘(’, expression, ‘)’ ;

[24] **Note:** no delimiter is allowed

A.7 Statements

```
statement = let statement
          | let be statement
          | def statement
          | block statement
          | assign statement
          | if statement
          | cases statement
          | sequence for loop
          | set for loop
          | index for loop
          | while loop
          | nondeterministic statement
          | call statement
          | specification statement
          | return statement
          | always statement
          | trap statement
          | recursive trap statement
          | exit statement
          | error statement                Non standard
          | identity statement ;
```

A.7.1 Local Binding Statements

```
let statement = 'let', local definition, { ',', local definition },
                'in', statement ;

local definition = value definition
                 | function definition ;

let be statement = 'let', bind, [ 'be', 'st', expression ], 'in',
                   statement ;

def statement = 'def', equals definition,
                { ';', equals definition }, [ ';' ],              Non standard
                'in', statement ;

equals definition = pattern bind, '=', expression ;
```

A.7.2 Block and Assignment Statements

block statement = ‘(’, { dcl statement },
statement, { ‘;’, statement }, [‘;’], ‘)’ ; Non standard

dcl statement = ‘dcl’, assignment definition,
{ ‘,’, assignment definition }, ‘;’ ;

assignment definition = identifier, ‘:’, type, [‘:=’, expression] ;

assign statement = state designator, ‘:=’, expression ;

multiple assign statement = ‘atomic’, ‘(’ assign statement, ‘;’,
assign statement,
[{ ‘;’, assign statement }], ‘)’ ;

A.7.3 Conditional Statements

if statement = ‘if’, expression, ‘then’, statement,
{ elseif statement },
[‘else’, statement] ;

elseif statement = ‘elseif’, expression, ‘then’, statement ;

cases statement = ‘cases’, expression, ‘:’,
cases statement alternatives,
[‘,’, others statement], ‘end’ ;

cases statement alternatives = cases statement alternative,
{ ‘,’, cases statement alternative } ;

cases statement alternative = pattern list, ‘->’, statement ;

others statement = ‘others’, ‘->’, statement ;

A.7.4 Loop Statements

sequence for loop = ‘for’, pattern bind, ‘in’, [‘reverse’],
expression, ‘do’, statement ;

set for loop = ‘for’, ‘all’, pattern, ‘in set’, expression,
‘do’, statement ;

index for loop = ‘for’, identifier, ‘=’, expression, ‘to’, expression,
[‘by’, expression],
‘do’, statement ;

while loop = ‘while’, expression, ‘do’, statement ;

A.7.5 The Nondeterministic Statement

nondeterministic statement = ‘||’, ‘(’, statement,
{ ‘,’, statement }, ‘)’ ;

A.7.6 Call and Return Statements

call statement = name, ‘(’,
[expression list], ‘)’ ;

return statement = ‘return’, [expression] ;

A.7.7 The Specification Statement

specification statement = ‘[’, implicit operation body, ‘]’ ; Non standard

A.7.8 Exception Handling Statements

always statement = ‘always’, statement, ‘in’, statement ;

trap statement = ‘trap’, pattern bind, ‘with’, statement,
‘in’, statement ;

recursive trap statement = 'tixe', traps, 'in', statement ;

traps = '{', pattern bind, '|->', statement,
{ ',', pattern bind, '|->', statement }, '}' ;

exit statement = 'exit', [expression] ;

A.7.9 The Error Statement

error statement = 'error' ; Non standard

A.7.10 The Identity Statement

identity statement = 'skip' ;

A.8 Patterns and Bindings

A.8.1 Patterns

pattern = pattern identifier
| match value
| set enum pattern
| set union pattern
| seq enum pattern
| seq conc pattern
| map enumeration pattern
| map muinon pattern
| tuple pattern
| record pattern ;

pattern identifier = identifier | '-' ;

match value = '(', expression, ')'
| symbolic literal ;

set enum pattern = '{', [pattern list], '}' ; Non standard

set union pattern = pattern, 'union', pattern ;

seq enum pattern = '[', [pattern list], ']' ; Non standard

seq conc pattern = pattern, '^', pattern ;

map enumeration pattern = '{', [maplet pattern list], '}' ;

maplet pattern list = maplet pattern, { ',', maplet pattern } ;

maplet pattern = pattern, '|->', pattern ;

map muinon pattern = pattern, 'munion', pattern ;

tuple pattern = 'mk_', '(', pattern, ',', pattern list, ')' ;

tuple pattern = 'mk_', '(', pattern, ',', pattern list, ')' ;

record pattern = 'mk_',[25] name, '(', [pattern list], ')' ;

pattern list = pattern, { ',', pattern } ;

[25]**Note:** no delimiter is allowed

A.8.2 Bindings

pattern bind = pattern | bind ;

bind = set bind | type bind ;

set bind = pattern, 'in set', expression ;

type bind = pattern, ':', type ;

bind list = multiple bind, { ',', multiple bind } ;

multiple bind = multiple set bind
| multiple type bind ;

multiple set bind = pattern list, 'in set', expression ;

multiple type bind = pattern list, ':', type ;

type bind list = type bind, { ',', type bind } ;

B Lexical Specification

B.1 Characters

The character set is shown in Table 11, with the forms of characters used in this document. Notice that this character set corresponds exactly to the ASCII (or ISO 646) syntax.

In the VDM-SL standard a character is defined as:

character = plain letter
| key word letter
| distinguished letter

| Greek letter
| digit
| delimiter character
| other characters
| separator ;

The plain letters and the keyword letters are displayed in Table 11 (in a document the keyword letters simply use the corresponding small letters). The distinguished letters use the corresponding capital and lower-case letters where the whole quote literal is preceded by "<" and followed by ">" (note that quote literals can also use underscores and digits). The Greek letters can also be used with a number sign "#" followed by the corresponding letter (this information is used by the LaTeX pretty printer such that the Greek letters can be produced). All delimiter characters (in the ASCII version of the standard) are listed in Table 11. In the standard a distinction between delimiter characters and compound delimiters are made. We have chosen not to use this distinction in this presentation. Please also notice that some of the delimiters in the mathematical syntax are keywords in the ASCII syntax which is used here.

plain letter:
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

keyword letter:
a b c d e f g h i j k l m
n o p q r s t u v w x y z

delimiter character:
```
,    :    ;    =    (    )    |    -    [    ]
{    }    +    /    <    >    <=   >=   <>   .
*    ->   +>   ==>  ||   =>   <=>  |->  <:   :>
<-:  :->  &    ==   **   ^    ++
```

digit:
0 1 2 3 4 5 6 7 8 9

hexadecimal digit:
0 1 2 3 4 5 6 7 8 9
A B C D E F
a b c d e f

octal digit:
0 1 2 3 4 5 6 7

other characters:
_ ‘ ’ " @ ~

newline:

white space:

These have no graphic form, but are a combination of white space and line break. There are two separators: without line break (white space) and with line break (newline).

Table 11: Character set

B.2 Symbols

The following kinds of symbols exist: keywords, delimiters, symbolic literals, and comments. The transformation from characters to symbols is given by the following rules; these use the same notation as the syntax definition but differ in meaning in that no separators may appear between adjacent terminals. Where ambiguity is possible otherwise, two consecutive symbols must be separated by a separator.

keyword = ‘abs’ | ‘all’ | ‘always’ | ‘and’ | ‘as’ | ‘atomic’ | ‘be’ | ‘bool’ | ‘by’
| ‘card’ | ‘cases’ | ‘char’ | ‘comp’ | ‘compose’ | ‘conc’
| ‘dcl’ | ‘def’ | ‘definitions’ | ‘dinter’ | ‘div’ | ‘dlmodule’
| ‘do’ | ‘dom’ | ‘dunion’ | ‘elems’ | ‘else’ | ‘elseif’ | ‘end’
| ‘error’ | ‘errs’ | ‘exists’ | ‘exists1’ | ‘exit’ | ‘exports’ | ‘ext’
| ‘false’ | ‘floor’ | ‘for’ | ‘forall’ | ‘from’ | ‘functions’
| ‘hd’ | ‘if’ | ‘imports’ | ‘in’ | ‘inds’ | ‘init’ | ‘inmap’
| ‘int’ | ‘inter’ | ‘inv’ | ‘inverse’ | ‘iota’
| ‘lambda’ | ‘len’ | ‘let’ | ‘map’ | ‘measure’ | ‘merge’ | ‘mod’ | ‘module’
| ‘mu’ | ‘munion’ | ‘nat’ | ‘nat1’ | ‘nil’ | ‘not’ | ‘of’
| ‘operations’ | ‘or’ | ‘others’ | ‘post’
| ‘power’ | ‘pre’ | ‘psubset’ | ‘rat’ | ‘rd’ | ‘real’ | ‘rem’
| ‘renamed’ | ‘return’ | ‘reverse’ | ‘rng’ | ‘seq’ | ‘seq1’
| ‘set’ | ‘skip’ | ‘specified’ | ‘st’ | ‘state’ | ‘struct’
| ‘subset’ | ‘then’ | ‘tixe’ | ‘tl’ | ‘to’ | ‘token’ | ‘trap’
| ‘true’ | ‘types’ | ‘undefined’ | ‘union’ | ‘uselib’
| ‘values’ | ‘while’ | ‘with’ | ‘wr’ | ‘yet’
| ‘RESULT’ ;

separator = newline | white space ;

identifier = (plain letter | Greek letter),
{ (plain letter | Greek letter) | digit | ‘’’ | ‘_’ } ;

All identifiers beginning with one of the reserved prefixes are reserved: init_, inv_, is_, mk_, post_ and pre_.

type variable identifier = ‘@’, identifier ;

is basic type = 'is_', ('bool' | 'nat' | 'nat1' | 'int' | 'rat'
| 'real' | 'char' | 'token') ;

symbolic literal = numeric literal | boolean literal
| nil literal | character literal | text literal
| quote literal ;

numeral = digit, { digit } ;

numeric literal = decimal literal | hexadecimal literal ;

exponent = ('E' | 'e'), ['+' | '-'], numeral ;

decimal literal = numeral, ['.', digit, { digit }], [exponent] ;

hexadecimal literal = ('0x' | '0X'), hexadecimal digit, { hexadecimal digit } ;

boolean literal = 'true' | 'false' ;

nil literal = 'nil' ;

character literal = ''', character | escape sequence
| multi character, ''' ; Non standard

escape sequence = '\\' | '\r' | '\n' | '\t' | '\f' | '\e' | '\a'
| '\x' hexadecimal digit,hexadecimal digit | '\c' character
| '\' octal digit, octal digit, octal digit
| '\"' | '\'' | ;

multi character = Greek letter
| '<=' | '>=' | '<>' | '->' | '+>' | '==>' | '||'
| '=>' | '<=>' | '|->' | '<:' | ':>' | '<-:'
| ':->' | '==' | '**' | '++' ;

text literal = '"', { '""' | character | escape sequence }, '"' ;

quote literal = distinguished letter,
{ '_' | distinguished letter | digit } ;

Single-line comment = '--', { character – newline }, newline ;

The escape sequences given above are to be interpreted as follows:

Sequence	Interpretation
'\\'	backslash character
'\r'	return character
'\n'	newline character
'\t'	tab character
'\f'	formfeed character
'\e'	escape character
'\a'	alarm (bell)
'\x' hexadecimal digit, hexadecimal digit	hex representation of character (e.g. \x41 is 'A')
'\c' character	control character (e.g. \c A ≡ \x01)
'\' octal digit, octal digit, octal digit	octal representation of character
'\"'	the " character
'\''	the ' character

C Operator Precedence

The precedence ordering for operators in the concrete syntax is defined using a two-level approach: operators are divided into families, and an upper-level precedence ordering, $>$, is given for the families, such that if families F_1 and F_2 satisfy

$$F_1 > F_2$$

then every operator in the family F_1 is of a higher precedence than every operator in the family F_2.

The relative precedences of the operators within families is determined by considering type information, and this is used to resolve ambiguity. The type constructors are treated separately, and are not placed in a precedence ordering with the other operators.

There are six families of operators, namely Combinators, Applicators, Evaluators, Relations, Connectives and Constructors:

Combinators: Operations that allow function and mapping values to be combined, and function, mapping and numeric values to be iterated.

Applicators: Function application, field selection, sequence indexing, etc.

Evaluators: Operators that are non-predicates.

Relations: Operators that are relations.

Connectives: The logical connectives.

Constructors: Operators that are used, implicitly or explicitly, in the construction of expressions; e.g. if-then-elseif-else, '|->', '...', etc.

The precedence ordering on the families is:

combinators > applicators > evaluators > relations > connectives > constructors

C.1 The Family of Combinators

These combinators have the highest family priority.

combinator = iterate | composition ;

iterate = '**' ;

composition = 'comp' ;

precedence level	combinator
1	comp
2	iterate

C.2 The Family of Applicators

All applicators have equal precedence.

applicator = subsequence
| apply
| function type instantiation
| field select ;

subsequence = expression, '(', expression, ',', '...', ',',
expression, ')' ;

apply = expression, '(', [expression list], ')' ;

function type instantiation = expression, '[', type, { ',', type }, ']' ;

field select = expression, '.', identifier ;

C.3 The Family of Evaluators

The family of evaluators is divided into nine groups, according to the type of expression they are used in.

evaluator = arithmetic prefix operator
  | set prefix operator
  | sequence prefix operator
  | map prefix operator
  | map inverse
  | arithmetic infix operator
  | set infix operator
  | sequence infix operator
  | map infix operator ;

arithmetic prefix operator = ‘+’ | ‘−’ | ‘abs’ | ‘floor’ ;

set prefix operator = ‘card’ | ‘power’ | ‘dunion’ | ‘dinter’ ;

sequence prefix operator = ‘hd’ | ‘tl’ | ‘len’
  | ‘inds’ | ‘elems’ | ‘conc’ ;

map prefix operator = ‘dom’ | ‘rng’ | ‘merge’ | ‘inverse’ ;

arithmetic infix operator = ‘+’ | ‘−’ | ‘*’ | ‘/’ | ‘rem’ | ‘mod’ | ‘div’ ;

set infix operator = ‘union’ | ‘inter’ | ‘\’ ;

sequence infix operator = ‘^’ ;

map infix operator = ‘munion’ | ‘++’ | ‘<:’ | ‘<-:’ | ‘:>’ | ‘:->’ ;

The precedence ordering follows a pattern of analogous operators. The family is defined in the following table.

precedence level	arithmetic	set	map	sequence
1	+ -	union \	munion ++	^
2	* /	inter		
	rem			
	mod			
	div			
3			inverse	
4			<: <-:	
5			:> :->	
6	(unary) +	card	dom	len
	(unary) -	power	rng	elems
	abs	dinter	merge	hd tl
	floor	dunion		conc
				inds

C.4 The Family of Relations

This family includes all the relational operators whose results are of type bool.

relation = relational infix operator | set relational operator ;

relational infix operator = ‘=’ | ‘<>’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ ;

set relational operator = ‘subset’ | ‘psubset’ | ‘in set’ | ‘not in set’ ;

precedence level	relation	
1	<=	<
	>=	>
	=	<>
	subset	psubset
	in set	not in set

All operators in the Relations family have equal precedence. Typing dictates that there is no meaningful way of using them adjacently.

C.5 The Family of Connectives

This family includes all the logical operators whose result is of type bool.

connective = logical prefix operator | logical infix operator ;

logical prefix operator = 'not' ;

logical infix operator = 'and' | 'or' | '=>' | '<=>' ;

precedence level	connective
1	<=>
2	=>
3	or
4	and
5	not

C.6 The Family of Constructors

This family includes all the operators used to construct a value. Their priority is given either by brackets, which are an implicit part of the operator, or by the syntax.

C.7 Grouping

The grouping of operands of the binary operators are as follows:

Combinators: Right grouping.

Applicators: Left grouping.

Connectives: The '=>' operator has right grouping. The other operators are associative and therefore right and left grouping are equivalent.

Evaluators: Left grouping[26].

Relations: No grouping, as it has no meaning.

Constructors: No grouping, as it has no meaning.

C.8 The Type Operators

Type operators have their own separate precedence ordering, as follows:

1. Function types: `->`, `+>` (right grouping).
2. Union type: `|` (left grouping).
3. Other binary type operators: `*` (no grouping).
4. Map types: `map` ... `to` ... and `inmap` ... `to` ... (right grouping). Non standard
5. Unary type operators: `seq of`, `seq1 of`, `set of`.

D Differences between the two Concrete Syntaxes

Below is a list of the symbols which are different in the mathematical syntax and the ASCII syntax:

Mathematical syntax	ASCII syntax
$\cdot$	`&`
$\times$	`*`
$\leq$	`<=`
$\geq$	`>=`
$\neq$	`<>`
$\xrightarrow{o}$	`==>`
$\rightarrow$	`->`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`

[26]Except the "map domain restrict to" and the "map domain restrict by" operators which have a right grouping. This is not standard.

Mathematical syntax	ASCII syntax
$\mapsto$	|->
$\underline{\triangle}$	==
$\uparrow$	**
$\dagger$	++
$\uplus$	munion
$\lhd$	<:
$\rhd$	:>
$\lhd\!\!\!-$	<-:
$-\!\!\!\rhd$	:->
$\subset$	psubset
$\subseteq$	subset
$\curvearrowright$	^
$\bigcap$	dinter
$\bigcup$	dunion
$\mathcal{F}$	power
$\ldots$-set	set of ...
$\ldots^*$	seq of ...
$\ldots^+$	seq1 of ...
$\ldots\xrightarrow{m}\ldots$	map ... to ...
$\ldots\xleftrightarrow{m}\ldots$	inmap ... to ...
μ	mu
$\mathbb{B}$	bool
$\mathbb{N}$	nat
$\mathbb{Z}$	int
$\mathbb{R}$	real
$\neg$	not
$\cap$	inter
$\cup$	union
$\in$	in set
$\notin$	not in set
$\wedge$	and
$\vee$	or
$\forall$	forall
$\exists$	exists
$\exists!$	exists1
λ	lambda
ι	iota
$\ldots^{-1}$	inverse ...

E Standard Libraries

E.1 Math Library

The Math library is defined in the `math.vdm` file. It provides the following math functions:

Functions		**Pre-conditions**
`sin: real +> real`	Sine	
`cos: real +> real`	Cosine	
`tan: real -> real`	Tangent	The argument is not an integer multiple of $\pi/2$
`cot: real -> real`	Cotagent	The argument is not an integer multiple of π
`asin: real -> real`	Inverse sine	The argument is not in the interval from -1 to 1 (both inclusive).
`acos: real -> real`	Inverse cosine	The argument is not in the interval from -1 to 1 (both inclusive).
`atan:real +> real`	Inverse tangent	
`sqrt: real -> real`	Square root	The argument is non-negative.

and the value:

`pi` = 3.14159265358979323846

If the functions are applied with arguments that violate possible pre-conditions they will return values that are not proper VDM-SL values, `Inf` (infinity, e.g. `tan(pi/2)`) and `NaN` (not a number, e.g. `sqrt (-1)`).

To use the standard library in a modular specification, the library file

```
$TOOLBOXHOME/stdlib/math.vdm
```

should be added to the current project. This contains the module `MATH`. Functions from this library may then be accessed in the usual way, by importing them into modules as needed. The example below demonstrates this:

```
module UseLib

  imports
    from MATH all

  definitions

  types

  coord :: x : real
           y : real

  functions

  -- euclidean metric between two points
  dist : coord * coord -> real
  dist (c1,c2) ==
    MATH`sqrt((c1.x - c2.x) * (c1.x - c2.x) +
              (c1.y - c2.y) * (c1.y - c2.y));


  -- outputs angle of line joining coord with origin
  -- from horizontal, in degrees
  angle : coord -> real
  angle (c) ==
    MATH`atan (c.y / c.x) * 360 / ( 2 * MATH`pi)

end UseLib
```

E.2 IO Library

The IO library is defined in the `io.vdm` file, and it is located in the directory `$TOOLBOXHOME/stdlib/`. It provides the IO functions and operations listed below. Each read/write function or operation returns a boolean value (or a tuple with a boolean component) representing the success (true) or failure (false) of the corresponding IO action.

`writeval[@p]:[@p] +> bool`
: This function writes a VDM value in ASCII format to standard output. There is no pre-condition.

fwriteval[@p]:seq1 of char * @p * filedirective +> bool

This function writes a VDM value (the second argument) in ASCII format to a file whose name is specified by the character string in the first argument. The third parameter has type filedirective which is defined to be:

```
filedirective = <start>|<append>
```

If <start> is used, the existing file (if any) is overwritten; if <append> is used, output is appended to the existing file and a new file is created if one does not already exist. There is no pre-condition.

freadval[@p]:seq1 of char +> bool * [@p]

This function reads a VDM value in ASCII format from the file specified by the character string in the first argument. There is no pre-condition. The function returns a pair, the first component indicating the success of the read and the second component indicating the value read if the read was successful.

echo: seq of char ==> bool

This operation writes the given text to standard output. Surrounding double quotes will be stripped, backslashed characters will be interpreted as escape sequences. There is no pre-condition.

fecho: seq of char * seq of char * [filedirective] ==> bool

This operation is similar to echo but writes text to a file rather than to standard output. The filedirective parameter should be interpreted as for fwriteval. The pre-condition for this operation is that if an empty string is given for the filename, then the [filedirective] argument should be nil since the text is written to standard output.

ferror:() ==> seq of char The read/write functions and operations return false if an error occurs. In this case an internal error string will be set. This operation returns this string and sets it to "".

As an example of the use of the IO library, consider a web server which maintains a log of page hits:

```
module LoggingWebServer

  imports
```

```
    from IO all

  exports all

  definitions

  values
    logfilename : seq1 of char = "serverlog"

  functions
    URLtoString : URL -> seq of char
    URLtoString = ...

  operations
    RetrieveURL : URL ==> File
    RetrieveURL(url) ==
      (def - = IO`fecho(logfilename, URLtoString(url)^"\n", <append>);
       ...
      );

    ResetLog : () ==> bool
    ResetLog() ==
      IO`fecho(logfilename,"\n",<start>)

end LoggingWebServer
```

E.3 VDMUtil Library

The VDMUtil library is defined in the `vdmutil.vdm` file, and it is located in the directory `$TOOLBOXHOME/stdlib/`. It provides the different kind of VDM utility functions and operations listed below.

`set2seq[@T]:set of @T +> seq of @T`
: This utility function enables an easy conversion of a set of elements without ordering into a sequence with an arbitrary ordering of the elements.

`get_file_pos: () +> [seq of char * nat * nat * seq of char * seq of char]`

: This function is able to extract context information (file name, line number, class name and function/operation name) for a particular part of the source text.

`val2seq_of_char[@T]: @T +> seq of char`
: This function is able to transform a VDM value into a string.

`seq_of_char2val[@p]:seq1 of char -> bool * [@p]`
: This function is able to transform a string (a sequence of chars) into a VDM value.

```
module VDMUtil

--  VDMTools STANDARD LIBRARY: VDMUtil
--      --------------------------------------------
--
-- Standard library for the VDMTools Interpreter. When the interpreter
-- evaluates the preliminary functions/operations in this file,
-- corresponding internal functions is called instead of issuing a run
-- time error. Signatures should not be changed, as well as name of
-- module (VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
-- Dont care's may NOT be used in the parameter lists.

exports all
definitions
functions
-- Converts a set argument into a sequence in non-deterministic order.
set2seq[@T] : set of @T +> seq of @T
set2seq(x) == is not yet specified;

-- Returns a context information tuple which represents
-- (file_name * line_num * column_num * module_name * fnop_name)
-- of corresponding source text
get_file_pos : () +> [ seq of char * nat * nat * seq of char * seq of char ]
get_file_pos() == is not yet specified;

-- Converts a VDM value into a seq of char.
val2seq_of_char[@T] : @T +> seq of char
val2seq_of_char(x) == is not yet specified;

-- converts VDM value in ASCII format into a VDM value
-- RESULT.#1 = false implies a conversion failure
```

```
seq_of_char2val[@p]:seq1 of char -> bool * [@p]
seq_of_char2val(s) ==
  is not yet specified
  post let mk_(b,t) = RESULT in not b => t = nil;

end VDMUtil
```

Index