

**Overture Technical Report Series  
No. TR-001**

**Feb 2018**

**VDM-10 Language Manual**

by

Peter Gorm Larsen  
Kenneth Lausdahl  
Nick Battle  
John Fitzgerald  
Sune Wolff  
Shin Sahara  
Marcel Verhoef  
Peter W. V. Tran-Jørgensen  
Tomohiro Oda  
Paul Chisholm



**Document history**

Month	Year	Version	Version of Overture.exe	Comment
April	2010		0.2	
May	2010	1	0.2	
February	2011	2	1.0.0	
July	2012	3	1.2.2	
April	2013	4	2.0.0	
March	2014	5	2.0.4	Includes RMs #16, #17, #18, #20
November	2014	6	2.1.2	Includes RMs #25, #26, #29
August	2015	7	2.3.0	Includes RMs #27
April	2016	8	2.3.4	Review inputs from Paul Chisholm
September	2016	9	2.4.0	RMs #35, #36
May	2017	10	2.5.0	RM #39
Feb	2018	11	2.6.0	Includes RM #42

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The VDM Specification Language (VDM-SL) . . . . .	1
1.2	The VDM++ Language . . . . .	1
1.3	The VDM Real Time Language (VDM-RT) . . . . .	2
1.4	Purpose of The Document . . . . .	2
1.5	Structure of the Document . . . . .	2
<b>2</b>	<b>Concrete Syntax Notation</b>	<b>3</b>
<b>3</b>	<b>Data Type Definitions</b>	<b>5</b>
3.1	Basic Data Types . . . . .	5
3.1.1	The Boolean Type . . . . .	6
3.1.2	The Numeric Types . . . . .	8
3.1.3	The Character Type . . . . .	11
3.1.4	The Quote Type . . . . .	11
3.1.5	The Token Type . . . . .	12
3.2	Compound Types . . . . .	12
3.2.1	Set Types . . . . .	13
3.2.2	Sequence Types . . . . .	15
3.2.3	Map Types . . . . .	18
3.2.4	Product Types . . . . .	22
3.2.5	Composite Types . . . . .	23
3.2.6	Union and Optional Types . . . . .	26
3.2.7	The Object Reference Type (VDM++ and VDM-RT) . . . . .	28
3.2.8	Function Types . . . . .	29
3.3	Invariants . . . . .	31
3.4	Equality . . . . .	32
3.5	Order . . . . .	35
<b>4</b>	<b>Algorithm Definitions</b>	<b>39</b>



<b>5</b>	<b>Function Definitions</b>	<b>41</b>
5.1	Polymorphic Functions . . . . .	47
5.2	Higher Order Functions . . . . .	49
<b>6</b>	<b>Expressions</b>	<b>51</b>
6.1	Let Expressions . . . . .	51
6.2	The Define Expression . . . . .	54
6.3	Unary and Binary Expressions . . . . .	55
6.4	Conditional Expressions . . . . .	56
6.5	Quantified Expressions . . . . .	58
6.6	The Iota Expression . . . . .	60
6.7	Set Expressions . . . . .	61
6.8	Sequence Expressions . . . . .	63
6.9	Map Expressions . . . . .	65
6.10	Tuple Constructor Expressions . . . . .	66
6.11	Record Expressions . . . . .	66
6.12	Apply Expressions . . . . .	68
6.13	The New Expression (VDM++ and VDM-RT) . . . . .	69
6.14	The Self Expression (VDM++ and VDM-RT) . . . . .	70
6.15	The Threadid Expression (VDM++ and VDM-RT) . . . . .	71
6.16	The Lambda Expression . . . . .	73
6.17	Narrow Expressions . . . . .	74
6.18	Is Expressions . . . . .	76
6.19	Base Class Membership (VDM++ and VDM-RT) . . . . .	77
6.20	Class Membership . . . . .	77
6.21	Same Base Class Membership (VDM++ and VDM-RT) . . . . .	78
6.22	Same Class Membership (VDM++ and VDM-RT) . . . . .	78
6.23	History Expressions (VDM++ and VDM-RT) . . . . .	79
6.24	The Time Expression (VDM-RT) . . . . .	80
6.25	Literals and Names . . . . .	81
6.26	The Undefined Expression . . . . .	83
6.27	The Precondition Expression . . . . .	83
<b>7</b>	<b>Patterns</b>	<b>87</b>
7.1	Object Pattern (VDM++ and VDM-RT) . . . . .	92
<b>8</b>	<b>Bindings</b>	<b>95</b>
<b>9</b>	<b>Value (Constant) Definitions</b>	<b>97</b>
<b>10</b>	<b>Declaration of Modifiable State Components</b>	<b>99</b>
10.1	Instance Variables (VDM++ and VDM-RT) . . . . .	99
10.2	The State Definition (VDM-SL) . . . . .	101



<b>11 Operation Definitions</b>	<b>103</b>
11.1 Constructors (VDM++ and VDM-RT)	110
<b>12 Statements</b>	<b>111</b>
12.1 Let Statements	111
12.2 The Define Statement	113
12.3 The Block Statement	114
12.4 The Assignment Statement	115
12.5 Conditional Statements	119
12.6 For-Loop Statements	121
12.7 The While-Loop Statement	123
12.8 The Nondeterministic Statement	124
12.9 The Call Statement	126
12.10 The Return Statement	129
12.11 Exception Handling Statements	129
12.12 The Error Statement	132
12.13 The Identity Statement	133
12.14 Start and Start List Statements (VDM++ and VDM-RT)	134
12.15 Stop and Stop List Statements (VDM++ and VDM-RT)	135
12.16 The Specification Statement	136
12.17 The Duration Statement (VDM-RT)	137
12.18 The Cycles Statement (VDM-RT)	139
<b>13 Top-level Specification in VDM</b>	<b>141</b>
13.1 Top-level Specification in VDM-SL	141
13.1.1 A Flat Specification	141
13.1.2 A Structured Specification	143
13.2 Top-level Specification in VDM++ and VDM-RT	150
13.3 System (VDM-RT)	150
13.3.1 Classes	153
13.3.2 Inheritance	155
13.3.3 Interface and Availability of Class Members	157
<b>14 Synchronization Constraints (VDM++ and VDM-RT)</b>	<b>163</b>
14.1 Permission Predicates	164
14.1.1 History guards	165
14.1.2 The object state guard	166
14.1.3 Queue condition guards	167
14.1.4 Evaluation of Guards	168
14.2 Inheritance of Synchronization Constraints	168
14.2.1 Mutex constraints	168



<b>15</b>	<b>Threads (VDM++ and VDM-RT)</b>	<b>171</b>
15.1	Periodic Thread Definitions (VDM-RT)	171
15.2	Sporadic Thread Definitions (VDM-RT)	174
15.3	Procedural Thread Definitions (VDM++ and VDM-RT)	175
<b>16</b>	<b>Trace Definitions</b>	<b>179</b>
<b>A</b>	<b>The Syntax of the VDM Languages</b>	<b>183</b>
A.1	VDM-SL Document	183
A.1.1	Modules	183
A.2	VDM++ and VDM-RT Document	185
A.3	System (VDM-RT)	185
A.3.1	Classes	185
A.4	Definitions	185
A.4.1	Type Definitions	185
A.4.2	The VDM-SL State Definition	187
A.4.3	Value Definitions	188
A.4.4	Function Definitions	188
A.4.5	Operation Definitions	189
A.4.6	Instance Variable Definitions (VDM++ and VDM-RT)	190
A.4.7	Synchronization Definitions (VDM++ and VDM-RT)	191
A.4.8	Thread Definitions (VDM++ and VDM-RT)	191
A.4.9	Trace Definitions	191
A.5	Expressions	192
A.5.1	Bracketed Expressions	193
A.5.2	Local Binding Expressions	194
A.5.3	Conditional Expressions	194
A.5.4	Unary Expressions	194
A.5.5	Binary Expressions	196
A.5.6	Quantified Expressions	198
A.5.7	The Iota Expression	198
A.5.8	Set Expressions	198
A.5.9	Sequence Expressions	198
A.5.10	Map Expressions	198
A.5.11	The Tuple Constructor Expression	199
A.5.12	Record Expressions	199
A.5.13	Apply Expressions	199
A.5.14	The Lambda Expression	199
A.5.15	The narrow Expression	199
A.5.16	The New Expression (VDM++ and VDM-RT)	199
A.5.17	The Self Expression (VDM++ and VDM-RT)	199
A.5.18	The Threadid Expression (VDM++ and VDM-RT)	200



A.5.19	The Is Expression . . . . .	200
A.5.20	The Undefined Expression . . . . .	200
A.5.21	The Precondition Expression . . . . .	200
A.5.22	Base Class Membership (VDM++ and VDM-RT) . . . . .	200
A.5.23	Class Membership (VDM++ and VDM-RT) . . . . .	200
A.5.24	Same Base Class Membership (VDM++ and VDM-RT) . . . . .	200
A.5.25	Same Class Membership (VDM++ and VDM-RT) . . . . .	200
A.5.26	History Expressions (VDM++ and VDM-RT) . . . . .	201
A.5.27	Time Expressions (VDM-RT) . . . . .	201
A.5.28	Names . . . . .	201
A.6	State Designators . . . . .	201
A.7	Statements . . . . .	201
A.7.1	Local Binding Statements . . . . .	202
A.7.2	Block and Assignment Statements . . . . .	202
A.7.3	Conditional Statements . . . . .	203
A.7.4	Loop Statements . . . . .	203
A.7.5	The Nondeterministic Statement . . . . .	204
A.7.6	Call and Return Statements . . . . .	204
A.7.7	The Specification Statement . . . . .	204
A.7.8	Start and Start List Statements (VDM++ and VDM-RT) . . . . .	204
A.7.9	Stop and Stop List Statements (VDM++ and VDM-RT) . . . . .	204
A.7.10	The Duration and Cycles Statements (VDM-RT) . . . . .	205
A.7.11	Exception Handling Statements . . . . .	205
A.7.12	The Error Statement . . . . .	205
A.7.13	The Identity Statement . . . . .	205
A.8	Patterns and Bindings . . . . .	205
A.8.1	Patterns . . . . .	205
A.8.2	Bindings . . . . .	207
<b>B</b>	<b>Lexical Specification</b>	<b>209</b>
B.1	Characters . . . . .	209
B.2	Symbols . . . . .	211
<b>C</b>	<b>Operator Precedence</b>	<b>215</b>
C.1	The Family of Combinators . . . . .	216
C.2	The Family of Applicators . . . . .	216
C.3	The Family of Evaluators . . . . .	216
C.4	The Family of Relations . . . . .	218
C.5	The Family of Connectives . . . . .	218
C.6	The Family of Constructors . . . . .	219
C.7	Grouping . . . . .	219
C.8	The Type Operators . . . . .	219



<b>D Differences between the Concrete Syntaxes</b>	<b>221</b>
--	------------





# Chapter 1

## Introduction

The Vienna Development Method (VDM) [?, ?, ?] was originally developed at the IBM laboratories in Vienna in the 1970's and as such it is one of the longest established formal methods. This document is a common language manual for the three dialects for VDM-SL, VDM++ and VDM-RT in the VDM-10 commonly agreed language revision. These dialects are supported by both VDMTools (minus VDM-RT) [?] (in the appropriate version) as well as in the Overture open source tool [?] built on top of the Eclipse platform. Whenever a construct is common to the three different dialects the term “VDM languages” will be used. Whenever a construct is specific to a subset of the VDM languages the specific dialect term mentioned above will be mentioned explicitly.

### 1.1 The VDM Specification Language (VDM-SL)

The syntax and semantics of the VDM-SL language is essentially the standard ISO/VDM-SL [?] with a modular extension<sup>1</sup>. 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<sup>2</sup>. Wherever the VDM-SL notation differs from the VDM-SL standard notation the semantics will of course be carefully explained.

### 1.2 The VDM++ Language

VDM++ is a formal specification language intended to specify object oriented systems with parallel and real-time behaviour, typically in technical environments [?]. The language is based on VDM-SL [?], and has been extended with class and object concepts, which are also present in languages like Smalltalk-80 and Java. This combination facilitates the development of object oriented formal specifications.

---

<sup>1</sup>A few other extensions are also included.

<sup>2</sup>A more tutorial like presentation is given in [?, ?] whereas proofs in VDM-SL are treated best in [?] and [?].



## 1.3 The VDM Real Time Language (VDM-RT)

The VDM-RT language (formerly called VICE as an acronym for “VDM++ In Constrained Environments”) is used to appropriately model and analyse Real-Time embedded and distributed systems [?, ?, ?, ?, ?]. Thus VDM-RT is an extension of the VDM++ language.

## 1.4 Purpose of The Document

This document is the language reference manual for all the VDM-10 dialects. The syntax of VDM language constructs is defined using grammar rules. The meaning of each language construct is explained in an informal manner and some small examples are given. The description is supposed to be suited for ‘looking up’ information rather than for ‘sequential reading’; it is a manual rather than a tutorial. The reader is expected to be familiar with the concepts of object oriented programming/design.

We will use the ASCII (also called the interchange) concrete syntax but we will display all reserved words in a special keyword font. Note that general Unicode identifiers are allowed so it is for example possible to write Japanese characters directly.

## 1.5 Structure of the Document

Chapter 2 presents the BNF notation used for the description of syntactic constructs. The VDM notations are described in Chapter 3 to Chapter 16. 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 differences between symbols in the mathematical syntax and the ASCII concrete syntax.

## Chapter 2

### 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.)



# Chapter 3

## Data Type Definitions

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

**types**

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 the VDM languages 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 the VDM languages.

In this chapter 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.

### 3.1 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.<sup>1</sup>

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.

### 3.1.1 The Boolean Type

In general the VDM languages allow one to specify systems in which computations may fail to terminate or to deliver a result. To deal with such potential undefinedness, the VDM languages employ a three valued logic: values may be true, false or bottom (undefined). The semantics of the VDM interpreters differs from the ISO/VDM-SL standard in that it does not have an LPF (Logic of Partial Functions) three valued logic where the order of the operands is unimportant (see [?]). 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 VDM interpreter can still be considered to be three-valued as for ISO/VDM-SL. However, bottom values may either result in infinite computation or a run-time error in the VDM interpreter.

**Name:** Boolean

**Symbol:** **bool**

**Values:** **true**, **false**

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

Operator	Name	Type
<b>not</b> <i>b</i>	Negation	<b>bool</b> → <b>bool</b>
<i>a</i> <b>and</b> <i>b</i>	Conjunction	<b>bool</b> * <b>bool</b> → <b>bool</b>
<i>a</i> <b>or</b> <i>b</i>	Disjunction	<b>bool</b> * <b>bool</b> → <b>bool</b>
<i>a</i> => <i>b</i>	Implication	<b>bool</b> * <b>bool</b> → <b>bool</b>
<i>a</i> <=> <i>b</i>	Biimplication	<b>bool</b> * <b>bool</b> → <b>bool</b>
<i>a</i> = <i>b</i>	Equality	<b>bool</b> * <b>bool</b> → <b>bool</b>
<i>a</i> <> <i>b</i>	Inequality	<b>bool</b> * <b>bool</b> → <b>bool</b>

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

<sup>1</sup>In these examples the Meta symbol '≡' will be used to indicate what the given example is equivalent to.



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<sup>2</sup>:

Negation <b>not</b> $b$	$b$	<b>true</b>	<b>false</b>	$\perp$
	<b>not</b> $b$	<b>false</b>	<b>true</b>	$\perp$

Conjunction $a$ <b>and</b> $b$	$a \backslash b$	<b>true</b>	<b>false</b>	$\perp$
	<b>true</b>	<b>true</b>	<b>false</b>	$\perp$
	<b>false</b>	<b>false</b>	<b>false</b>	<b>false</b>
	$\perp$	$\perp$	$\perp$	$\perp$

Disjunction $a$ <b>or</b> $b$	$a \backslash b$	<b>true</b>	<b>false</b>	$\perp$
	<b>true</b>	<b>true</b>	<b>true</b>	<b>true</b>
	<b>false</b>	<b>true</b>	<b>false</b>	$\perp$
	$\perp$	$\perp$	$\perp$	$\perp$

Implication $a \Rightarrow b$	$a \backslash b$	<b>true</b>	<b>false</b>	$\perp$
	<b>true</b>	<b>true</b>	<b>false</b>	$\perp$
	<b>false</b>	<b>true</b>	<b>true</b>	<b>true</b>
	$\perp$	$\perp$	$\perp$	$\perp$

Biimplication $a \Leftrightarrow b$	$a \backslash b$	<b>true</b>	<b>false</b>	$\perp$
	<b>true</b>	<b>true</b>	<b>false</b>	$\perp$
	<b>false</b>	<b>false</b>	<b>true</b>	$\perp$
	$\perp$	$\perp$	$\perp$	$\perp$

**Examples:** Let  $a = \text{true}$  and  $b = \text{false}$  then:

<b>not</b> $a$	$\equiv$	<b>false</b>
$a$ <b>and</b> $b$	$\equiv$	<b>false</b>
$b$ <b>and</b> $\perp$	$\equiv$	<b>false</b>
$a$ <b>or</b> $b$	$\equiv$	<b>true</b>
$a$ <b>or</b> $\perp$	$\equiv$	<b>true</b>
$a \Rightarrow b$	$\equiv$	<b>false</b>
$b \Rightarrow b$	$\equiv$	<b>true</b>
$b \Rightarrow \perp$	$\equiv$	<b>true</b>
$a \Leftrightarrow b$	$\equiv$	<b>false</b>
$a = b$	$\equiv$	<b>false</b>
$a <> b$	$\equiv$	<b>true</b>
$\perp$ <b>or not</b> $\perp$	$\equiv$	$\perp$
$(b \text{ and } \perp) \text{ or } (\perp \text{ and false})$	$\equiv$	$\perp$

<sup>2</sup>Notice that in standard VDM-SL all these truth tables (except  $\Rightarrow$ ) would be symmetric.





### 3.1.2 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 subset hierarchy where **real** is the most general type followed by **rat**<sup>3</sup>, **int**, **nat** and **nat1**. Note that no “casting” like it is done in many programming languages is needed in the VDM languages.

Type	Values
<b>nat1</b>	1, 2, 3, ...
<b>nat</b>	0, 1, 2, ...
<b>int</b>	..., -2, -1, 0, 1, ...
<b>real</b>	..., -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	<b>real, rat, int, nat, nat1</b>
3.0	<b>real, rat, int, nat, nat1</b>
0	<b>real, rat, int, nat</b>
-1	<b>real, rat, int</b>
3.1415	<b>real, rat</b>

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.

<sup>3</sup>From the VDM interpreter’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	<b>real</b> $\rightarrow$ <b>real</b>
<b>abs</b> $x$	Absolute value	<b>real</b> $\rightarrow$ <b>real</b>
<b>floor</b> $x$	Floor	<b>real</b> $\rightarrow$ <b>int</b>
$x + y$	Sum	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
$x - y$	Difference	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
$x * y$	Product	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
$x / y$	Division	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
$x \text{ div } y$	Integer division	<b>int</b> * <b>int</b> $\rightarrow$ <b>int</b>
$x \text{ rem } y$	Remainder	<b>int</b> * <b>int</b> $\rightarrow$ <b>int</b>
$x \text{ mod } y$	Modulus	<b>int</b> * <b>int</b> $\rightarrow$ <b>int</b>
$x ** y$	Power	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
$x < y$	Less than	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
$x > y$	Greater than	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
$x \leq y$	Less or equal	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
$x \geq y$	Greater or equal	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
$x = y$	Equal	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
$x <> y$	Not equal	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>

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 $x \geq 0$ 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  $-14 \text{ **div** } 3$ : either (the intuitive)  $-4$  as in the VDM interpreters, or  $-5$  as in e.g. Standard ML [?]. It is therefore appropriate to explain these operations in some detail.

Integer division is defined using **floor** and real number division:

$$\begin{aligned} x/y < 0: \quad & x \text{ **div** } y = -\text{floor}(\text{abs}(-x/y)) \\ x/y \geq 0: \quad & x \text{ **div** } y = \text{floor}(\text{abs}(x/y)) \end{aligned}$$



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:

$$\begin{aligned} x \text{ rem } y &= x - y * (x \text{ div } y) \\ x \text{ mod } y &= x - y * \text{floor}(x/y) \end{aligned}$$

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 visits, because the second argument corresponding to  $y$  is positive.

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

$-a$	$\equiv$	$-7$
<b>abs</b> $a$	$\equiv$	$7$
<b>abs</b> $d$	$\equiv$	$3$
<b>floor</b> $a \leq a$	$\equiv$	<b>true</b>
$a + d$	$\equiv$	$4$
$a * b$	$\equiv$	$24.5$
$a / b$	$\equiv$	$2$
$a \text{ div } e$	$\equiv$	$3$
$a \text{ div } d$	$\equiv$	$-2$
$a \text{ mod } e$	$\equiv$	$1$
$a \text{ mod } d$	$\equiv$	$-2$
$-a \text{ mod } d$	$\equiv$	$-1$
$a \text{ rem } e$	$\equiv$	$1$
$a \text{ rem } d$	$\equiv$	$1$
$-a \text{ rem } d$	$\equiv$	$-1$
$3**2 + 4**2 = 5**2$	$\equiv$	<b>true</b>
$b < c$	$\equiv$	<b>false</b>
$b > c$	$\equiv$	<b>true</b>
$a \leq d$	$\equiv$	<b>false</b>
$b \geq e$	$\equiv$	<b>true</b>
$a = e$	$\equiv$	<b>false</b>
$a = 7.0$	$\equiv$	<b>true</b>
$c <> d$	$\equiv$	<b>true</b>
<b>abs</b> $c < 0$	$\equiv$	<b>false</b>



$$(a \text{ **div** } e) * e \quad \equiv \quad 6$$

### 3.1.3 The Character Type

The character type contains all the single character elements of the VDM character set (see Table B.1 on page 210).

**Name:** Char

**Symbol:** **char**

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

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

Operator	Name	Type
<i>c1</i> = <i>c2</i>	Equal	<b>char</b> * <b>char</b> → <b>bool</b>
<i>c1</i> <> <i>c2</i>	Not equal	<b>char</b> * <b>char</b> → <b>bool</b>

**Examples:**

```
'a' = 'b'    ≡ false
'1' = 'c'    ≡ false
'd' <> '7'    ≡ true
'e' = 'e'    ≡ true
```

### 3.1.4 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 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
<i>q</i> = <i>r</i>	Equal	<b>T</b> * <b>T</b> → <b>bool</b>
<i>q</i> <> <i>r</i>	Not equal	<b>T</b> * <b>T</b> → <b>bool</b>



**Examples:** Let  $T$  be the type defined as:

$T = \langle \text{France} \rangle \mid \langle \text{Denmark} \rangle \mid \langle \text{SouthAfrica} \rangle \mid \langle \text{SaudiArabia} \rangle$

If for example  $a = \langle \text{France} \rangle$  then:

$\langle \text{France} \rangle = \langle \text{Denmark} \rangle \quad \equiv \quad \mathbf{false}$   
 $\langle \text{SaudiArabia} \rangle \langle \rangle \langle \text{SouthAfrica} \rangle \quad \equiv \quad \mathbf{true}$   
 $a \langle \rangle \langle \text{France} \rangle \quad \equiv \quad \mathbf{false}$

### 3.1.5 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, 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 that contain token types. However, in order to resemble the ISO/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	<b>token</b> * <b>token</b> $\rightarrow$ <b>bool</b>
$s \langle \rangle t$	Not equal	<b>token</b> * <b>token</b> $\rightarrow$ <b>bool</b>

**Examples:** Let for example  $s = \mathbf{mk\_token}(6)$ , let  $t = \mathbf{mk\_token}(1)$  and  $u = \mathbf{mk\_token}(\{1, 2\})$  in:

$s = t \quad \equiv \quad \mathbf{false}$   
 $s \langle \rangle t \quad \equiv \quad \mathbf{true}$   
 $s = \mathbf{mk\_token}(6) \quad \equiv \quad \mathbf{true}$   
 $u = \mathbf{mk\_token}(\{2, 1\}) \quad \equiv \quad \mathbf{true}$

## 3.2 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<sup>4</sup>.
- 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.

### 3.2.1 Set Types

A set is an unordered collection of values, all of the same type<sup>5</sup>, which is treated as a whole. All sets in VDM languages 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. Sets may include the empty set (`set0 type`), or may require at least one element (`set1 type`).

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  $e_n$  are elements from the sets,  $bd1$ ,  $bd2$ , ...,  $bd_m$  are bindings of identifiers to sets or types, and  $P$  is a logical predicate.

**Syntax:**    `type = set type`  
               |    `... ;`

`set type = set0 type`  
               |    `set1 type ;`

`set0 type = 'set of', type ;`

`set1 type = 'set1 of', type ;`

**Equation:**  $S = \text{set of } A$  or  $S = \text{set1 of } A$

**Constructors:**

**Set enumeration:**  $\{e1, e2, \dots, e_n\}$  constructs a set of the enumerated elements.  
 The empty set is denoted by  $\{\}$ .

<sup>4</sup>These operators are used in either unary or binary expressions which are given with all the operators in section 6.3.

<sup>5</sup>Note however that it is always possible to find a common type for two values by the use of a union type (see section 3.2.6.)



**Set comprehension:**  $\{e \mid bd1, bd2, \dots, bdm \ \& \ P\}$  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, a sequence binding, or a type binding<sup>6</sup>. A set bind  $bdn$  has the form  $pat1, \dots, patp$  **in set**  $s$ , where  $pati$  is a pattern (normally simply an identifier), and  $s$  is a set constructed by an expression. A sequence (or type) binding is similar, in the sense that **in set** is replaced by **in seq** (or a colon) and  $s$  is replaced with a sequence (or type) expression.

The syntax and semantics for all set expressions are given in section 6.7.

### Operators:

Operator	Name	Type
$e$ <b>in set</b> $s1$	Membership	$A * \text{set of } A \rightarrow \text{bool}$
$e$ <b>not in set</b> $s1$	Not membership	$A * \text{set of } A \rightarrow \text{bool}$
$s1$ <b>union</b> $s2$	Union	$\text{set of } A * \text{set of } A \rightarrow \text{set of } A$
$s1$ <b>inter</b> $s2$	Intersection	$\text{set of } A * \text{set of } A \rightarrow \text{set of } A$
$s1 \setminus s2$	Difference	$\text{set of } A * \text{set of } A \rightarrow \text{set of } A$
$s1$ <b>subset</b> $s2$	Subset	$\text{set of } A * \text{set of } A \rightarrow \text{bool}$
$s1$ <b>psubset</b> $s2$	Proper subset	$\text{set of } A * \text{set of } A \rightarrow \text{bool}$
$s1 = s2$	Equality	$\text{set of } A * \text{set of } A \rightarrow \text{bool}$
$s1 <> s2$	Inequality	$\text{set of } A * \text{set of } A \rightarrow \text{bool}$
<b>card</b> $s1$	Cardinality	$\text{set of } A \rightarrow \text{nat}$
<b>dunion</b> $ss$	Distributed union	$\text{set of set of } A \rightarrow \text{set of } A$
<b>dinter</b> $ss$	Distributed intersection	$\text{set1 of set of } A \rightarrow \text{set of } A$
<b>power</b> $s1$	Finite power set	$\text{set of } A \rightarrow \text{set of set of } A$

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 3.2.6.

### 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 $s2$ .

<sup>6</sup>Notice that type bindings over infinite types (discharging the invariant limitations) cannot be executed by the VDM interpreters because in general they are not executable (see section 8 for further information about this).



Operator Name	Semantics Description
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 $s2 \setminus s1$ 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 $ss$ .
Distributes intersection	the resulting set is the intersection of all the elements (these are sets themselves) of $ss$ , 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  $s1 = \{\langle \text{France} \rangle, \langle \text{Denmark} \rangle, \langle \text{SouthAfrica} \rangle, \langle \text{SaudiArabia} \rangle\}$ ,  $s2 = \{2, 4, 6, 8, 11\}$  and  $s3 = \{\}$  then:

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

### 3.2.2 Sequence Types

A sequence value is an ordered collection of elements of some type indexed by  $1, 2, \dots, n$ ; 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,  $l1$  is a sequence of sequence values.  $e1$ ,  $e2$  and  $e_n$  are elements in these sequences,  $i$  will be a natural number,  $P$  is a predicate and  $e$  is an arbitrary expression.

**Syntax:**     $\text{type} = \text{seq type}$   
                    $\mid \dots ;$

$\text{seq type} = \text{seq0 type}$   
                    $\mid \text{seq1 type} ;$

$\text{seq0 type} = \text{'seq of', type} ;$

$\text{seq1 type} = \text{'seq1 of', type} ;$

**Equation:**  $L = \text{seq of } A$  or  $L = \text{seq1 of } A$

**Constructors:**

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

**Sequence comprehension:**  $[e \mid id \text{ in seq } S \ \& \ P]$  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 sequence of elements and  $id$  will be matched with the elements preserving the order of  $S$ .

$[e \mid id \text{ in set } S \ \& \ P]$  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 elements of a type with an order relation (section 3.5) and  $id$  will be matched to the elements from the set in increasing order.

The syntax and semantics of all sequence expressions are given in section 6.8.

**Operators:**



Operator	Name	Type
<b>hd</b> $l$	Head	<b>seq</b> of $A \rightarrow A$
<b>tl</b> $l$	Tail	<b>seq</b> of $A \rightarrow$ <b>seq</b> of $A$
<b>len</b> $l$	Length	<b>seq</b> of $A \rightarrow$ <b>nat</b>
<b>elems</b> $l$	Elements	<b>seq</b> of $A \rightarrow$ <b>set</b> of $A$
<b>inds</b> $l$	Indexes	<b>seq</b> of $A \rightarrow$ <b>set</b> of <b>nat</b>
<b>reverse</b> $l$	Reverse	<b>seq</b> of $A \rightarrow$ <b>seq</b> of $A$
$l1 \wedge l2$	Concatenation	$(\text{seq of } A) * (\text{seq of } A) \rightarrow \text{seq of } A$
<b>conc</b> $l1$	Distributed concatenation	<b>seq</b> of <b>seq</b> of $A \rightarrow$ <b>seq</b> of $A$
$l ++ m$	Sequence modification	<b>seq</b> of $A * \text{map nat1 to } A \rightarrow \text{seq of } A$
$l(i)$	Sequence application	<b>seq</b> of $A * \text{nat1} \rightarrow A$
$l1 = l2$	Equality	$(\text{seq of } A) * (\text{seq of } A) \rightarrow \text{bool}$
$l1 <> l2$	Inequality	$(\text{seq of } A) * (\text{seq of } A) \rightarrow \text{bool}$

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 3.2.6.

### 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 $\{1, \dots, \text{len } l\}$ .
Reverse	yields a new sequence where the order of the elements have been reversed.
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 $l1$ 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. $\text{dom } m$ must be a subset of $\text{inds } l$
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:

<b>len</b> $l1$	$\equiv$	7
<b>hd</b> $(l1^{\wedge}l2)$	$\equiv$	3
<b>tl</b> $(l1^{\wedge}l2)$	$\equiv$	$[1, 4, 1, 5, 9, 2, 2, 7, 1, 8]$
$l3(\mathbf{len} \ l3)$	$\equiv$	$<Tunisia>$
<b>"England"</b> (2)	$\equiv$	'n'
<b>reverse</b> $l1$	$\equiv$	$[2, 9, 5, 1, 4, 1, 3]$
<b>conc</b> $[l1, l2] = l1^{\wedge}l2$	$\equiv$	<b>true</b>
<b>conc</b> $[l1, l1, l2] = l1^{\wedge}l2$	$\equiv$	<b>false</b>
<b>elems</b> $l3$	$\equiv$	$\{<England>, <Rumania>, <Colombia>, <Tunisia>\}$
<b>(elems</b> $l1)$ <b>inter</b> <b>(elems</b> $l2)$	$\equiv$	$\{1, 2\}$
<b>inds</b> $l1$	$\equiv$	$\{1, 2, 3, 4, 5, 6, 7\}$
<b>(inds</b> $l1)$ <b>inter</b> <b>(inds</b> $l2)$	$\equiv$	$\{1, 2, 3, 4\}$
$l3 \ ++ \ \{2 \ \mapsto \ <Germany>, 4 \ \mapsto \ <Nigeria>\}$	$\equiv$	$[<England>, <Germany>, <Colombia>, <Nigeria>]$

### 3.2.3 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 languages 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:**     $\text{type} = \text{map type}$   
                   |    ... ;

map type = general map type



| injective map type ;

general map type = **'map'**, type, **'to'**, type ;

injective map type = **'inmap'**, type, **'to'**, type ;

**Equation:**  $M = \text{map } A \text{ to } B$  or  $M = \text{inmap } A \text{ to } B$

### Constructors:

**Map enumeration:**  $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \dots, a_n \mapsto b_n\}$  constructs a mapping of the enumerated maplets. The empty map is written as  $\{\mapsto\}$ .

**Map comprehension:**  $\{ed \mapsto er \mid bd_1, \dots, bd_n \ \& \ P\}$  constructs a mapping by evaluating the expressions  $ed$  and  $er$  on all the possible bindings for which the predicate  $P$  evaluates to **true**.  $bd_1, \dots, bd_n$  are bindings of free identifiers from the expressions  $ed$  and  $er$  to sets, sequences or types.

The syntax and semantics of all map expressions are given in section 6.9.

### Operators:

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

**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.



Operator Name	Semantics Description
Merge	yields a map combined by $m_1$ and $m_2$ such that the resulting map maps the elements of <b>dom</b> $m_1$ as does $m_1$ , and the elements of <b>dom</b> $m_2$ as does $m_2$ . The two maps must be compatible.
Override	overrides and merges $m_1$ with $m_2$ , i.e. it is like a merge except that $m_1$ and $m_2$ need not be compatible; any common elements are mapped as by $m_2$ (so $m_2$ overrides $m_1$ ).
Distributed merge	yields the map that is constructed by merging all the maps in $m_s$ . The maps in $m_s$ must be compatible.
Domain restrict to	creates the map consisting of the elements in $m$ whose key is in $s$ . $s$ need not be a subset of <b>dom</b> $m$ .
Domain restrict by	creates the map consisting of the elements in $m$ whose key is not in $s$ . $s$ need not be a subset of <b>dom</b> $m$ .
Range restrict to	creates the map consisting of the elements in $m$ whose information value is in $s$ . $s$ need not be a subset of <b>rng</b> $m$ .
Range restrict by	creates the map consisting of the elements in $m$ whose information value is not in $s$ . $s$ need not be a subset of <b>rng</b> $m$ .
Map apply	yields the information value whose key is $d$ . $d$ must be in the domain of $m$ .
Map composition	yields the map that is created by composing $m_2$ elements with $m_1$ elements. The resulting map is a map with the same domain as $m_2$ . The information value corresponding to a key is the one found by first applying $m_2$ to the key and then applying $m_1$ to the result. <b>rng</b> $m_2$ must be a subset of <b>dom</b> $m_1$ .
Map iteration	yields the map where $m$ is composed with itself $n$ times. $n=0$ yields the identity map where each element of <b>dom</b> $m$ is map into itself; $n=1$ yields $m$ itself. For $n>1$ , the range of $m$ must be a subset of <b>dom</b> $m$ .
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,
      <England> |-> 4}                   ≡ {<France> |-> 8,
                                           <Denmark> |-> 4,
                                           <SouthAfrica> |-> 2,
                                           <SaudiArabia> |-> 1,
                                           <England> |-> 4}

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

```



$$\begin{aligned}
\text{inverse } m2 &\equiv \{2 \mid\rightarrow 1, 3 \mid\rightarrow 2, \\
&\quad 4 \mid\rightarrow 3, 1 \mid\rightarrow 4 \} \\
m2 \text{ comp } (\text{inverse } m2) &\equiv \{1 \mid\rightarrow 1, 2 \mid\rightarrow 2, \\
&\quad 3 \mid\rightarrow 3, 4 \mid\rightarrow 4 \}
\end{aligned}$$

### 3.2.4 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:**  $T = A1 * A2 * \dots * An$

**Constructors:** The tuple constructor:  $\mathbf{mk\_}(a1, a2, \dots, an)$

The syntax and semantics for the tuple constructor are given in section 6.10.

**Operators:**

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

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 6.12.

**Examples:** Let  $a = \mathbf{mk\_}(1, 4, 8), b = \mathbf{mk\_}(2, 4, 8)$  then:

$$\begin{aligned}
a = b &\equiv \mathbf{false} \\
a <> b &\equiv \mathbf{true} \\
a = \mathbf{mk\_}(2, 4) &\equiv \mathbf{false}
\end{aligned}$$



### 3.2.5 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 languages, **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 3.2.6. 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, *A*<sub>1</sub>, . . . , *A*<sub>*m*</sub> are arbitrary types, *r*, *r*<sub>1</sub>, and *r*<sub>2</sub> are record values, *i*<sub>1</sub>, . . . , *i*<sub>*m*</sub> are selectors from the *r* record value (and these must be unique entrances inside one record definition), *e*<sub>1</sub>, . . . , *e*<sub>*m*</sub> 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 :: selffirst : A1
    selsec    : A2
```

or





```
A :: selffirst : 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.

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 section 3.2.6) 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 distinguish the alternatives.

**Constructors:** The record constructor: **mk**<sub>A</sub>(*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 6.11.

### Operators:

Operator	Name	Type
<code>r.i</code>	Field select	$A * Id \rightarrow A_i$
<code>r1 = r2</code>	Equality	$A * A \rightarrow \mathbf{bool}$
<code>r1 &lt;&gt; r2</code>	Inequality	$A * A \rightarrow \mathbf{bool}$
<b>is</b> <sub>A</sub> ( <i>r1</i> )	Is	$Id * \text{MasterA} \rightarrow \mathbf{bool}$

### Semantics of Operators:



Operator Name	Semantics Description
Field select	yields the value of the field with fieldname <i>i</i> in the record value <i>r</i> . <i>r</i> must have a field with name <i>i</i> .
Equality <sup>7</sup>	Structural equality over the record. That is, field-by-field equality, recursively applying equality to the constituent fields.

**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
```

<sup>7</sup>This equality is implicitly provided with the type. It is possible to override the primitive equality – see Section 3.4



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.

### 3.2.6 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:**  $B = A_1 \mid A_2 \mid \dots \mid [A_n]$

**Constructors:** None.

**Operators:**

Operator	Name	Type
$t_1 = t_2$	Equality	$A * A \rightarrow \mathbf{bool}$
$t_1 <> t_2$	Inequality	$A * A \rightarrow \mathbf{bool}$

**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
expr.altn.tp=nil    ≡ true

```

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>.

### 3.2.7 The Object Reference Type (VDM++ and VDM-RT)

The object reference type has been added as part of the standard VDM-SL types. Therefore there is no direct way of restricting the use of object reference types (and thus of objects) in a way that conforms to pure object oriented principles; no additional structuring mechanisms than classes are foreseen. From these principles it follows that the use of an object reference type in combination with a type constructor (record, map, set, etc.) should be treated with caution.

A value of the object reference type can be regarded as a *reference* to an object. If, for example, an instance variable (see section 10.1) is defined to be of this type, this makes the class in which that instance variable is defined, a ‘client’ of the class in the object reference type; a *clientship relation* is established between the two classes.

An object reference type is denoted by a class name. The class name in the object reference type must be the name of a class defined in the specification.

The only operators defined for values of this type is the test for equality (‘=’) and inequality (‘<>’). Equality is based on references rather than values. That is, if o1 and o2 are two distinct objects which happen to have the same contents, o1 = o2 will yield false.

**Constructors** Object references are constructed using the new expression (see section 6.13).

#### Operators

Operator	Name	Type
t1 = t2	Equality	A * A → <b>bool</b>
t1 <> t2	Inequality	A * A → <b>bool</b>

**Examples** An example of the use of object references is in the definition of the class of binary trees:

```

class Tree

types

    protected tree = <Empty> | node;

    public node :: lt    : Tree
                  nval  : int
                  rt    : Tree

instance variables

    protected root: tree := <Empty>;

```



```
end Tree
```

Here we define the type of nodes, which consist of a node value, and references to left and right tree objects. Details of access specifiers may be found in section 13.3.3.

### 3.2.8 Function Types

In the VDM languages function types can also be used in type definitions. A function type from a type A (actually a list of types as a tuple type) 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 5. 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:**  $F = A \rightarrow B$ <sup>8</sup> or  $F = A \twoheadrightarrow B$

**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 6.16.

<sup>8</sup>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
$f(a_1, \dots, a_n)$	Function apply	$A_1 * \dots * A_n \rightarrow B$
$f_1 \text{ comp } f_2$	Function composition	$(B \rightarrow C) * (A \rightarrow B) \rightarrow (A \rightarrow C)$
$f ** n$	Function iteration	$(A \rightarrow A) * \text{nat} \rightarrow (A \rightarrow A)$
$t_1 = t_2$	Equality	$A * A \rightarrow \text{bool}$
$t_1 <> t_2$	Inequality	$A * A \rightarrow \text{bool}$

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

**Semantics of Operators:**

Operator Name	Semantics Description
Function apply	yields the result of applying the function $f$ to the values of $a_j$ . See the definition of apply expressions in Section 6.12.
Function composition	it yields the function equivalent to applying first $f_2$ and then applying $f_1$ to the result. $f_1$ , but not $f_2$ may be Curried.
Function iteration	yields the function 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
incl = 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
incl = inc2     ≡ false
```



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

### 3.3 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;
```

Note that where the `compose` form is used to define a composite type with an invariant, a distinction must be drawn. Consider:





```
Range = compose Rng of low:nat high:nat end  
      inv mk_Rng(1, h) == 1 <= h;
```

This defines two types, `Range` and `Rng`, where `Rng` is a pair of natural numbers with no constraint, while `Range` is structurally the same as `Rng` but must additionally satisfy the invariant. For example, the following definitions are valid

```
r:Range = mk_Rng(1, 2);  
r2:Rng = mk_Rng(2, 1);  
r3 = mk_Rng(2, 1);
```

`r2` and `r3` need not satisfy the invariant (in the absence of explicit declaration, `r3` is of type `Rng`). On the other hand,

```
r4:Range = mk_Rng(2, 1);
```

is invalid since `r4` of type `Range` violates the invariant.

### 3.4 Equality

Every type defined in VDM, both basic and compound types, is provided with an equality relation by default as described earlier. The primitive equality is not always that which is desired. If the values of a data type are normalised then structural equality is adequate, but this is not always the case. Consider for example a data type that represents times and includes time zones. The same point in time is represented differently in different time zones. A type definition allows an equality relation to be defined explicitly for a type. In such a case the explicit equality relation is employed when comparing values of the type in preference to the primitive equality. The general scheme for defining an equality relation is:

```
Id = Type  
    eq pat1 = pat2 == expr
```

or

```
Id :: fields  
    eq pat1 = pat2 == expr
```

`pat1` and `pat2` are patterns for two values of the type (or composite type), and `expr` is a boolean expression that is true exactly when the expressions represented by `pat1` and `pat2` are equal.



When defined, the explicit equality relation is also employed for inequality comparison with  $<>$ . If an **eq** clause is defined, a new (total) function is created implicitly with the signature:

```
eq_T : T * T -> bool
```

such that  $\text{eq\_T}(t1, t2)$  denotes the same value as  $t1 = t2$ .

**Examples:** Flight matching

```
Flight :: id          : seq1 of char
        departure     : seq1 of char
        depTime       : DateTime
        destination:  seq1 of char
eq mk_Flight(i1,d1,dt1,a1) = mk_Flight(i2,d2,dt2,a2) ==
    i1 = i2 and d1 = d2 and a1 = a2 and
    within(dt1, dt2, mk_Minute(10));
```

A simplified definition of a flight consisting of an identifier, departure location, departure date/time, and destination location. Two records refer to the same flight if they have the same identifier, same departure location, same destination location, and a departure time within 10 minutes of each other; it is the last item that renders structural equality inadequate.<sup>9</sup> Given

```
f1 = mk_Flight("QF5", "YSSY", '17-04-01 12:20', "WSSS")
f2 = mk_Flight("QF5", "YSSY", '17-04-01 12:28', "WSSS")
f3 = mk_Flight("VOZ42", "YSSY", '16-12-25 02:21', "YBBN")
f4 = mk_Flight("VOZ42", "YSSY", '16-12-24 02:21', "YBBN")
f5 = mk_Flight("VOZ42", "YSSY", '16-12-24 02:21', "YMML")
```

We have

```
f1 = f2           ≡ true
f3 = f4           ≡ false
f1 = f3           ≡ false
f2 <> f4          ≡ true
eq_Flight(f4, f5) ≡ false
```

A proof obligation is a condition that needs to be satisfied to verify a specification is consistent. Whenever an equality relation is defined, the proof obligation requires the relation to be an equivalence relation; that is, it is reflexive, symmetric and transitive<sup>10</sup>. For a type  $T$  we have:

<sup>9</sup>We assume a type `DateTime`, a type `Minute`, and a function `within`.

<sup>10</sup>[https://en.wikipedia.org/wiki/Equivalence\\_relation](https://en.wikipedia.org/wiki/Equivalence_relation) gives a quick introduction to equivalence relation.



Reflexive:

```
forall x:T & x = x
```

Symmetric:

```
forall x,y:T & x = y => y = x
```

Transitive:

```
forall x,y,z:T & x = y and y = z => x = z
```

The equality relation employed when evaluating an expression depends on the type that is determined for the expression statically; i.e. when type checking occurs, not during expression evaluation. Consider the type definition

```
NATPAIR = nat * nat  
eq e1 = e2 == e1.#1 = e2.#1;
```

and the value definitions

```
x1:NATPAIR = mk_(1,2);  
x2:NATPAIR = mk_(1,3);  
x3          = mk_(1,2);  
x4          = mk_(1,3);
```

The following expressions all evaluate to true.

```
x1 = x2  
x1 = mk_(1,42)  
x1 <> mk_(2,2)  
x1 = x3  
x1 = x4  
x3 <> x4  
x3 = mk_(1,2)
```

Note for example  $x1 = x2$  because statically they are of type `NATPAIR` whose equality relation states they are equal if their first elements are equal. On the other hand  $x3 \neq x4$  because statically they are of type `nat * nat` whose implicit equality requires both the



first and second elements to be equal.  $x1 = x4$  is true because statically  $x4$  is interpreted as type `NATPAIR` due to the explicit declaration of `x1 : NATPAIR`.

### 3.5 Order

Numeric types (section 3.1.2) have a primitive order relation. An order relation can be defined explicitly for other types as part of the type definition. The general scheme for defining an order (strict less than) relation is:

```
Id = Type
  ord pat1 < pat2 == expr
```

or

```
Id :: fields
  ord pat1 < pat2 == expr
```

`pat1` and `pat2` are patterns for two values of the type (or composite type), and `expr` is a boolean expression that is true exactly when the expression represented by `pat1` is less than the expression represented by `pat2` in the required order relation.

If an **ord** clause is defined, three new functions are created implicitly with the signatures:

```
ord_T : T * T +> bool
max_T : T * T +> T
min_T : T * T +> T
```

such that

```
ord_T(t1,t2)  ≡ t1 < t2
max_T(t1,t2)  ≡ if ord_T(t1,t2) or t1 = t2 then t2 else t1
min_T(t1, t2) ≡ if ord_T(t1, t2) or t1 = t2 then t1 else t2
```

If an **ord** clause is defined for a type, then the infix operators `<`, `<=`, `>` and `>=` can be employed with expressions of that type.

The equality relation for a type is defined (either explicitly or implicitly), and if the order relation for a type is also defined (explicitly), we have

```
x <= y  <=> x < y or x = y
x > y   <=> not (x = y or x < y)
x >= y  <=> x = y or not x < y
```

The proof obligation for an order relation is that it be a strict partial order; that is, it is irreflexive and transitive<sup>11</sup>.

<sup>11</sup>[https://en.wikipedia.org/wiki/Partially\\_ordered\\_set](https://en.wikipedia.org/wiki/Partially_ordered_set) gives a quick introduction to ordering relation.



Irreflexive:

```
forall x:T & not x < x
```

Transitive:

```
forall x,y,z:T & x < y and y < z => x < z
```

The conditions relating <, <=, > and >= stated above are only guaranteed to hold for an order relation that satisfies the proof obligation. Note a strict partial order must be asymmetric

```
forall x,y:T & x < y => not y < x
```

However, asymmetry is derivable from irreflexivity and transitivity.

**Example:** Score revisited

```
Score :: team    : Team
      won       : nat
      drawn    : nat
      lost     : nat
      points   : nat
inv sc == sc.points = 3 * sc.won + sc.drawn
ord mk_Score(t1,w1,-,-,p1) < mk_Score(t2,w2,-,-,p2) ==
    p1 < p2 or
    p1 = p2 and w1 < w2 or
    p1 = p2 and w1 = w2 and t1 < t2;
```

In this case the order is as might be presented in a league table (with greatest element at top):

- Most points first;
- If equal on points, most wins first;
- Otherwise alphabetic ordering of team name (not defined here).

Given

```
sc1 = mk_Score (<France>, 2, 2, 0, 8);
sc2 = mk_Score (<Scotland>, 3, 0, 0, 9);
sc3 = mk_Score (<SouthAfrica>, 0, 3, 0, 3);
sc4 = mk_Score (<SaudiArabia>, 1, 0, 2, 3);
```



We have

```
sc1 < sc2           ≡ true
sc1 <= sc3          ≡ false
sc2 > sc3           ≡ true
sc4 >= sc3          ≡ true
sc4 < sc3           ≡ false
ord_Score(sc1, sc2) ≡ true
```

The type argument of a polymorphic function can depend on an order relation. For example, the following specifies a function that determines if a sequence is ordered:

```
ascending[@a]: seq of @a -> bool
ascending(s) ==
  forall i in set {1,...,len s - 1} & s(i) <= s(i+1);
```

The specification compares for order values of the type argument @a. There is no way to express that @a admits an order relation so the function is partial. A type error would be thrown at runtime if the function was executed with a type that does not admit an order relation (such as **bool**).



# Chapter 4

## Algorithm Definitions

In the VDM languages 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 the VDM languages 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 access to the global variables in any way.

However, in order to be able to execute both functions and operations by the VDM interpreters they must be defined explicitly<sup>1</sup>. In the VDM languages it is also possible for 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**.

---

<sup>1</sup>Implicitly specified functions and operations cannot in general be executed because their 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.





# Chapter 5

## Function Definitions

In the VDM languages 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 VDM++ and VDM-RT functions are automatically available in a static form (i.e. without having an instance of the defining class). Thus there is no need to use the **static** keyword that can be used for operations in VDM++ and VDM-RT. Functions are executed atomically - which is consistent with them being able to read instance variables of objects passed, and there being no sync clauses (see Chapter 14) for functions. In general, the syntax for the definition of a function is:

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

```
access function definition = [ access ], function definition ;
```

```
access = 'public'  
        | 'private'  
        | 'protected' ;
```

```
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', measure body ] ;
```



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

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

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

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'**  
| **'is subclass responsibility'** ;

measure body = expression  
| **'is not yet specified'** ;



Here **is not yet specified** may be used as the function body during development of a model; whereas the **is subclass responsibility** indicates that implementation of this body must be undertaken by any subclasses so that can only be used in VDM++ and VDM-RT.

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);
```

Note that implicitly functions are considered total. Thus if they are exported in VDM-SL they need to be exported using the total function type.

A simple example of an extended explicit function definition 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 VDM interpreters, 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 VDM interpreters 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 be defined using recursion (i.e. by calling themselves). When recursion is used one is recommended to add a ‘**measure**’ clause that can be used in the proof obligations generated from the model such that termination proofs can be carried out. A measure clause has one of two forms:

1. The identifier of a function whose argument type is that of the function being defined and result type is **nat**.
2. An expression of type **nat** that may include free instances of the argument variables in the function definition.

A slightly more general form is described later.

Consider the definition of the factorial function:

```
fac: nat +> nat
fac(n) ==
  if n = 0
  then 1
```



```
else n*fac(n-1);
```

A **measure** clause (of the first type) could be added by appending

```
measure id
```

where

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

giving

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

The proof obligation becomes

```
forall n:nat &  
  (not (n=0) =>  
    id(n) > id(n-1));
```

Verifying the proof obligation ensures the recursive function will terminate; the value of the measure function is smaller on each recursive call, and being of type **nat** the value cannot go below 0.

The second form of measure clause employs an expression of type **nat**

```
measure n
```

Note in this case the *n* in the measure clause is the argument variable *n* in the definition of *fac*. The full definition is

```
fac: nat +> nat  
fac(n) ==  
  if n = 0  
  then 1
```



```

    else n*fac(n-1)
measure n;

```

An example on sequences

```

dupe[@a]: seq of @a +> seq of @a
dupe(s) == cases s:
    [] -> [],
    [x] -> [x,x],
    t^u -> dupe t ^ dupe u
end
measure len s;

```

As before the *s* in the measure clause is the argument variable in the definition of *dupe*. The measure function reduces because the semantics of concatenation patterns ensure that the sequence *s* is split into *t* and *u* such that *t* and *u* contain at least one element each.

A measure clause is recommended for recursive functions, but not mandatory. VDM implementations typically issue a warning if a recursive function is defined without a measure clause. If one explicitly does not want to provide a measure clause and avoid a warning the **is not yet specified** expression can be used.

```

fac: nat +> nat
fac(n) ==
    if n = 0
    then 1
    else n*fac(n-1)
measure is not yet specified;

```

In some situations a measure clause that evaluates to a single **nat** is not adequate. More generally a measure clause can be a function that returns a **nat** product, or an expression whose type is a **nat** product. Consider the Ackermann function

```

ackermann: nat * nat +> nat
ackermann(m,n) == cases mk_(m,n) :
    mk_(0,q) -> q+1,
    mk_(p,0) -> ackermann(p-1,1),
    mk_(p,q) -> ackermann(p-1,ackermann(p,q-1))
end;

```

Note that in the recursive calls either the first or the second argument reduces, but not both. The measure clause is



```
measure mk_(m, n)
```

Order is lexicographic ordering on the **nat** tuple: in each recursive call either the first argument reduces, or the first argument is unchanged and the second argument reduces. Generally an arbitrary tuple can be expressed as the value in a measure clause.

Defining a measure for a function  $f$  with signature  $f : T_1 * \dots * T_n \rightarrow T$  causes creation of a function `measure_f` with signature `measure_f : T_1 * ... * T_n +> nat` or `measure_f : T_1 * ... * T_n +> nat * ... * nat`.

## 5.1 Polymorphic Functions

Functions can also be polymorphic in VDM. 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:<sup>1</sup>

```
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 6.12):

```
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) ==
```

<sup>1</sup>The examples for polymorphic functions are taken from [?]. 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.





```
m ++ { e |-> num_bag[@elem] (e, m) + 1 }
```

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
```

Finally, VDM makes no assumptions about the type bound to a type parameter. This binding must be made explicitly, either by making an assertion as part of the pre-condition of the function, or by making a test before the part of the function body that restricts the type parameter. For example, consider the function given below:

```
public f[@T]: seq of @T -> seq of @T
f(s) ==
  if is_(s, seq of real) then
    [r + 1 | r in seq s]
  else
    reverse s;
```

Here the type parameter `@T` denotes the element type of the sequence `s` passed to the function. Note how the body of the function assumes the type parameter to be defined for the `+` operator. This only works since the `if`-clause explicitly states that the parameter `s` is of type `seq of real`. Without the `if`-clause use of the `+` operator would fail due to `@T` not being of a numeric type. Alternatively, the assumption could be stated using a pre-condition e.g. using `is real`. This puts a restriction on the type parameter that applies to the entire body of the function, though with simple examples like this the effect is the same as explicitly declaring the type of the parameter in the function signature.

Polymorphic functions can also be recursive and in those cases it also makes sense to include a measure function. For example:

```
dlen[@A]: seq of seq of @A -> nat
dlen(l) ==
  if l = []
  then 0
```



```

    else len hd l + dlen[@A] (tl l)
measure Len;

Len[@A]: seq of seq of @A -> nat
Len(l) == len l;

```

where a proof obligation ensuring termination of this recursive function as:

```

(forall l:seq of (seq of (@A)) & ((not (l = [])) =>
(Len[@A] (l) > Len[@A] (tl l))))

```

## 5.2 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) ==
  [n | n in seq ns & p(n)];

```

Then `nat_filter (lambda x:nat & x mod 2 = 0, [1,2,3,4,5])`  $\equiv$  `[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) ==
  [i | i in seq l & p(i)];

```

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

Since the `fmap` function is recursive, it ought to have a **measure** function defined. In the case of curried functions, the measure function's parameters are the same as a de-curried version of the recursive function's parameters. For the `fmap` example, this would be:

```
m[@elem]: (@elem -> @elem) * seq of @elem -> nat
m(-, l) == len l;
```

Note that the measure function is also polymorphic, and must have the same type parameters as the function it measures. The proof obligation will also be polymorphic:

```
(forall f:(@elem -> @elem), l:seq of (@elem) &
  (not (l = [])) =>
    m[@elem](f, l) > m[@elem](f, (tl l))))
```

# Chapter 6

## Expressions

In this chapter 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.

### 6.1 Let Expressions

**Syntax:**    expression = let expression  
                              | let be expression  
                              | ... ;

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

let be expression = **'let'**, multiple 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 5.

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

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



where  $p_1, \dots, p_n$  are patterns,  $e_1, \dots, e_n$  are expressions which match the corresponding pattern  $p_i$ , and  $e$  is an expression, of any type, involving the pattern identifiers of  $p_1, \dots, p_n$ . It denotes the value of the expression  $e$  in the context in which the patterns  $p_1, \dots, p_n$  are matched against the corresponding expressions  $e_1, \dots, e_n$ .

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 the VDM languages this is not supported by the VDM interpreters. 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 mb be st e1 in e2
```

where  $mb$  is a multi-binding of one or more patterns (mostly just one pattern) to a set value (or a sequence or a type),  $e_1$  is a boolean expression, and  $e_2$  is an expression, of any type, involving the pattern identifiers of the patterns from  $mb$ . The **be st**  $e_1$  part is optional. The expression denotes the value of the expression  $e_2$  in the context in which all the patterns from  $mb$  have been matched against either an element in the set from  $mb$ , or an element in the sequence from  $mb$ , or against a value from the type in  $mb$ <sup>1</sup>. If the **st**  $e_1$  expression is present, only such bindings where  $e_1$  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 44:

```
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)
```

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

```
let mk_Score(-,w1,-,-,p1) = sc1,
```

<sup>1</sup>Remember that only the set and sequence bindings can be executed by means of the VDM interpreters.



```
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 (see page 87) 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 (see page 49) 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 6.7):

```
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 seq l be st
        forall x in seq 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.

## 6.2 The Define Expression

This expression can only be used inside operations which will be described in section 11. 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)  $pb1, \dots, pbn$  are matched against the corresponding expressions  $e1, \dots, en$ <sup>2</sup>.

<sup>2</sup>If binds are used, it simply means that the values which can match the pattern are further constrained by the type, sequence, or set expression as explained in Chapter 7.



**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 12.1 also gives an example of a *define expression*.

## 6.3 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' | 'reverse'  
                   | 'card' | 'power' | 'dunion' | 'dinter'  
                   | 'hd' | 'tl' | 'len' | 'elems' | 'inds' | 'conc'  
                   | 'dom' | 'rng' | 'merge' ;

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 Chapter 3, 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 Chapter 3, so none will be provided here.

## 6.4 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  $e_1$  is a boolean expression, while  $e_2$  and  $e_3$  are expressions of any type. The **if** expression denotes the value of  $e_2$  evaluated in the given context if  $e_1$  evaluates to true in the given context. Otherwise the **if** expression denotes the value of  $e_3$  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  $p_{ij}$ 's are patterns which are matched one by one against the expression  $e$ . The  $e_i$ '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  $e_i$  expression evaluated in the context in which one of the  $p_{ij}$  patterns has been matched against  $e$ . The chosen  $e_i$  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 the VDM languages corresponds to what is used in most programming languages, while the **cases** expression in the VDM languages 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.

## 6.5 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*, *pi in seq 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, a sequence 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 [?], 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 forall y in set s & x <= y
```

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
```

## 6.6 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, a sequence 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.

## 6.7 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:

$\{e_1, e_2, e_3, \dots, e_n\}$

where  $e_1$  up to  $e_n$  are general expressions. It constructs a set of the values of the enumerated expressions. The empty set is written as  $\{\}$ .

The *set comprehension* expression has the form:

$\{e \mid mbd_1, mbd_2, \dots, mbd_n \ \& \ P\}$

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 set bindings, sequence bindings, and type bindings. Thus  $mbd_n$  will look like  $pat_1$  **in set**  $s_1$ ,  $pat_2$  :  $tp_1$ ,  $pat_3$  **in seq**  $q_1$ , ... **in set**  $s_2$ , where  $pat_i$  is a pattern (normally simply an identifier),  $s_1$  and  $s_2$  are sets constructed by expressions, and  $q_1$  is a sequence constructed by an expression (whereas  $tp_1$  is used to illustrate that type binds can also be used). Notice however that type binds can only be executed by the VDM interpreters in case the types can be statically declared as finite.

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

$\{e_1, \dots, e_2\}$

where  $e_1$  and  $e_2$  are numeric expressions. The set range expression denotes the set of integers from  $e_1$  to  $e_2$  inclusive. If  $e_2$  is smaller than  $e_1$  the set range expression denotes the empty set.

**Examples:** Using the values  $Europe = \{<France>, <England>, <Denmark>, <Spain>\}$  and  $GroupC = \{sc_1, sc_2, sc_3, sc_4\}$  (where  $sc_1, \dots, sc_4$  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}

```

## 6.8 Sequence Expressions

**Syntax:** expression = ...

- | sequence enumeration
- | sequence comprehension
- | subsequence
- | ... ;

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

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

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

**Semantics:** A *sequence enumeration* has the form:

$[e_1, e_2, \dots, e_n]$

where  $e_1$  through  $e_n$  are general expressions. It constructs a sequence of the enumerated elements. The empty sequence is written as  $[]$ .

A *sequence comprehension* over sequences has the form:

$[e \mid \text{pat} \text{ **in seq** } S \ \& \ P]$

where the expression  $e$  will use the identifiers from the pattern  $\text{pat}$  (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 sequence of values. It constructs a sequence by evaluating the expression  $e$  on all the bindings for which the predicate  $P$  evaluates to true, preserving the order of elements in  $S$ .

A *sequence comprehension* over sets has the form:

$[e \mid \text{pat} \text{ in set } S \ \& \ P]$

where the expression  $e$  will use the identifiers from the pattern  $\text{pat}$  (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. The bindings of the pattern identifier must be to a type that admits an order relation, which dictates the ordering of the elements in the resulting sequence. It constructs a sequence by evaluating the expression  $e$  on all the (ordered) bindings for which the predicate  $P$  evaluates to **true**. Note it is not the result sequence that is ordered, but the sequence of values of the pattern identifier.

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:

$l(n1, \dots, n2)$

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:



```

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

```

## 6.9 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:

$$\{d_1 \mid\rightarrow r_1, d_2 \mid\rightarrow r_2, \dots, d_n \mid\rightarrow r_n\}$$

where all the domain expressions  $d_i$  and range expressions  $r_i$  are general expressions and all  $d_i$ 's must be different unless they point to the same value. The empty map is written as  $\{\mid\rightarrow\}$ .

A *map comprehension* has the form:

$$\{e_d \mid\rightarrow e_r \mid mbd_1, \dots, mbd_n \ \& \ P\}$$

where constructs  $mbd_1, \dots, mbd_n$  are multiple bindings of variables from the expressions  $e_d$  and  $e_r$  to sets (or types). The *map comprehension* constructs a mapping by evaluating the expressions  $e_d$  and  $e_r$  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:

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

## 6.10 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  $e_i$  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
```

## 6.11 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  $(e1, e2, \dots, en)$  matches the type of the corresponding entrances in the composite type  $T$ . Note that the reason why a *name* (and not an *identifier*) is used here is to take into account that one would like to be able to refer also to a class or a module where the record type is defined (see Chapter 13) this would look like  $mk\_MC\ T(e1, e2, \dots, en)$  where  $MC$  will be the name of a *module* or a *class*.

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}

```

## 6.12 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 3.2.5 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 3.2.3 it has already been shown what it means to look up in a map. Similarly in section 3.2.2 it is illustrated how indexing in a sequence is performed. Function calls are using a call by value semantics meaning that the values are passed as arguments. The only exception to this is in the VDM++ and VDM-RT dialects where object references are passed as call by reference but since the functions cannot adjust any instance variable this make no semantic difference.

In the VDM languages 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  $\mathbf{mk\_}(v_1, \dots, v_N)$  and  $M$  is an integer in the range  $\{1, \dots, N\}$  then  $e.\#M$  yields  $v_M$ . If  $M$  lies outside  $\{1, \dots, N\}$  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  $t_1, \dots, t_n$  are types. The resulting function uses the types  $t_1, \dots, t_n$  instead of the variable type names given in the function definition.

**Examples:** Recall that `GroupA` is a sequence (see page 64), `GroupG` is a map (see page 66) and `selection_sort` is a function (see page 54):

```
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 5:

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

≡

{ -1 |-> 1 }
```

### 6.13 The New Expression (VDM++ and VDM-RT)

**Syntax:** expression = ...  
                  | new expression ;

new expression = 'new', name, '(', [ expression list ], ')';

**Semantics:** The *new expression* has the form:



```
new classname(e1, e2, ..., en)
```

An object can be created (also called *instantiated*) from its class description using a *new expression*. The effect of a *new expression* is that a ‘new’, unique object as described in class `classname` is created. The value of the *new expression* is a reference to the new object.

If the *new expression* is invoked with no parameters, an object is created in which all instance variables take their “default” values (i.e. the values defined by their initialisation conditions). With parameters, the *new expression* represents a *constructor* (see Section 11.1) and creates customised instances (i.e. where the instance variables may take values which are different from their default values).

**Examples:** Suppose we have a class called `Queue` and that default instances of `Queue` are empty. Suppose also that this class contains a constructor (which will also be called `Queue`) which takes a single parameter which is a list of values representing an arbitrary starting queue. Then we can create default instances of `Queue` in which the actual queue is empty using the expression

```
new Queue()
```

and an instance of `Queue` in which the actual queue is, say, `e1, e2, e3` using the expression

```
new Queue([e1, e2, e3])
```

Using the class `Tree` defined on page 29 we create new `Tree` instances to construct nodes:

```
mk_node(new Tree(), x, new Tree())
```

## 6.14 The Self Expression (VDM++ and VDM-RT)

**Syntax:**    expression = ...  
                  | self expression ;

self expression = ‘**self**’ ;

**Semantics:** The *self expression* has the form:

```
self
```



The self expression returns a reference to the object currently being executed. It can be used to simplify the name space in chains of inheritance.

**Examples:** Using the class `Tree` defined on page 29 we can specify a subclass called `BST` which stores data using the binary search tree approach. We can then specify an operation which performs a binary search tree insertion:

```
Insert : int ==> ()
Insert (x) ==
  (dcl curr_node : Tree := self;

  while not curr_node.isEmpty() do
    if curr_node.rootval() < x
    then curr_node := curr_node.rightBranch()
    else curr_node := curr_node.leftBranch();
  curr_node.addRoot(x);
  )
```

This operation uses a self expression to find the root at which to begin traversal prior to insertion. Further examples are given in section 12.9.

## 6.15 The Threadid Expression (VDM++ and VDM-RT)

**Syntax:** expression = ...  
                  | threadid expression ;

threadid expression = **'threadid'** ;

**Semantics:** The *threadid expression* has the form:

**threadid**

The threadid expression returns a natural number which uniquely identifies the thread in which the expression is executed. Note that periodic threads get a new threadid at the start of each new period.

**Examples:** Using **threadid**'s it is possible to provide a VDM++ base class that implements a Java-style wait-notify in VDM++ using permission predicates. Any object that should be available for the wait-notify mechanism must derive from this base class.

```
class WaitNotify
```





```
instance variables
  waitset : set of nat := {};

operations
  protected wait: () ==> ()
  wait() ==
    let p = threadid
    in (
      AddToWaitSet( p );
      Awake();
    );

  AddToWaitSet : nat ==> ()
  AddToWaitSet( p ) ==
    waitset := waitset union { p };

  Awake: () ==> ()
  Awake() ==
    skip;

  protected notify: () ==> ()
  notify() ==
    if waitset <> {}
    then let arbitrary_process in set waitset
      in waitset := waitset \ {arbitrary_process};

  protected notifyAll: () ==> ()
  notifyAll() ==
    waitset := {};

sync
  mutex(notifyAll, AddToWaitSet, notify);
  per Awake => threadid not in set waitset;

end WaitNotify
```

In this example the **threadid** expression is used in two places:

- In the `Wait` operation for threads to register interest in this object.
- In the permission predicate for `Awake`. An interested thread should call `Awake` following registration using `Wait`. It will then be blocked until its `threadid` is removed



from the waitset following another thread's call to `notify`.

## 6.16 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 chapter 5. A function defined by a lambda expression can be Curried by using a new nested lambda expression in the body. 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 53:



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

## 6.17 Narrow Expressions

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

narrow expression = 'narrow\_', '(', expression, ',', type, ')';

**Semantics:** The *narrow expression* converts the given *expression* value into the given *type*, returning a value of that type. It is legal to downcast a class to one of its subclasses, and it is legal to narrow an expression of a union type to one of its subtypes. However, a conversions between two completely unrelated types is a type error. Note that a narrow expression does not guarantee that its argument will be of the correct type at runtime, but using narrow gives extra type information to the specification.

**Examples:** In following examples, the Test() and Test'() operations should give the same results, but there is a type error in Test() which is resolved in Test' using a narrow expression.

```
class S  
end S  
  
class C1 is subclass of S  
  
instance variables  
public a : nat := 1;  
  
end C1  
  
class C2 is subclass of S  
  
instance variables  
public b : nat := 2;  
  
end C2  
  
class A
```



```

operations
public
Test: () ==> seq of nat
Test() ==
  let list : seq of S = [ new C1(), new C2() ]
  in
    return [ let e = list(i)
              in cases true:
                (isofclass(C1, e)) -> e.a,
                (isofclass(C2, e)) -> e.b
              end | i in set inds list ];

public
Test': () ==> seq of nat
Test'() ==
  let list : seq of S = [ new C1(), new C2() ]
  in
    return [ let e = list(i)
              in cases true:
                (isofclass(C1, e)) -> narrow_(e, C1).a,
                (isofclass(C2, e)) -> narrow_(e, C2).b
              end | i in set inds list ];

end A

```

```

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

```

## 6.18 Is Expressions

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

general is expression = is expression  
                           | type judgement ;

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

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 cannot 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.

## 6.19 Base Class Membership (VDM++ and VDM-RT)

**Syntax:** expression = ...  
                   | isofbaseclass expression  
                   | ... ;

isofbaseclass expression = 'isofbaseclass', '(', name, ',', expression, ')';

**Semantic:** The function **isofbaseclass** when applied to an object reference *expression* and a class name *name* yields the boolean value **true** if and only if *name* is a root super-class in the inheritance chain of the object referenced to by *expression*, and **false** otherwise.

**Examples:** Suppose that `BinarySearchTree` is a subclass of `Tree`, `Tree` is not a subclass of any other class and `Queue` is not related by inheritance to either `Tree` or `BinarySearchTree`. Let *t* be an instance of `Tree`, *b* is an instance of `BinarySearchTree` and *q* is an instance of `Queue`. Then:

<b>isofbaseclass</b> ( <code>Tree</code> , <i>t</i> )	≡	<b>true</b>
<b>isofbaseclass</b> ( <code>BinarySearchTree</code> , <i>b</i> )	≡	<b>false</b>
<b>isofbaseclass</b> ( <code>Queue</code> , <i>q</i> )	≡	<b>true</b>
<b>isofbaseclass</b> ( <code>Tree</code> , <i>b</i> )	≡	<b>true</b>
<b>isofbaseclass</b> ( <code>Tree</code> , <i>q</i> )	≡	<b>false</b>

## 6.20 Class Membership

**Syntax** expression = ...  
                   | isofclass expression  
                   | ... ;

isofclass expression = 'isofclass', '(', name, ',', expression, ')';

**Semantics:** The function **isofclass** when applied to an object reference *expression* and a class name *name* yields the boolean value **true** if and only if *expression* refers to an object of class *name* or to an object of any of the subclasses of *name*, and **false** otherwise.



**Examples:** Assuming the classes `Tree`, `BinarySearchTree`, `Queue`, and identifiers `t`, `b`, `q` as in the previous example, we have:

```

isofclass (Tree, t)           ≡ true
isofclass (Tree, b)           ≡ true
isofclass (Tree, q)           ≡ false
isofclass (Queue, q)          ≡ true
isofclass (BinarySearchTree, t) ≡ false
isofclass (BinarySearchTree, b) ≡ true

```

## 6.21 Same Base Class Membership (VDM++ and VDM-RT)

**Syntax:** `expression = ...`  
                   | `samebaseclass expression`  
                   | `... ;`

`samebaseclass expression = 'samebaseclass',  
                                   '(', expression, ',', expression, ')';`

**Semantics:** The function **samebaseclass** when applied to object references `expression1` and `expression2` yields the boolean value **true** if and only if the objects denoted by `expression1` and `expression2` are instances of classes that can be derived from the same root superclass, and **false** otherwise.

**Examples:** Assuming the classes `Tree`, `BinarySearchTree`, `Queue`, and identifiers `t`, `b`, `q` as in the previous example, suppose that `AVLTree` is another subclass of `Tree`, `BalancedBST` is a subclass of `BinarySearchTree`, `a` is an instance of `AVLTree` and `bb` is an instance of `BalancedBST`:

```

samebaseclass (a, b)    ≡ true
samebaseclass (a, bb)   ≡ true
samebaseclass (b, bb)   ≡ true
samebaseclass (t, bb)   ≡ true
samebaseclass (q, a)    ≡ false

```

## 6.22 Same Class Membership (VDM++ and VDM-RT)

**Syntax:** `expression = ...`  
                   | `sameclass expression`  
                   | `... ;`

`sameclass expression = 'sameclass',  
                           '(', expression, ',', expression, ')';`



**Semantics:** The function **sameclass** when applied to object references *expression1* and *expression2* yields the boolean value **true** if and only if the objects denoted by *expression1* and *expression2* are instances of the same class, and **false** otherwise.

**Examples:** Assuming the classes *Tree*, *BinarySearchTree*, *Queue*, and identifiers *t*, *b*, *q* from section 6.19, and assuming *b'* is another instance of *BinarySearchTree* we have:

```
sameclass(b,t)    ≡  false
sameclass(b,b')  ≡  true
sameclass(q,t)    ≡  false
```

## 6.23 History Expressions (VDM++ and VDM-RT)

**Syntax:** expression = ...  
                   | act expression  
                   | fin expression  
                   | active expression  
                   | req expression  
                   | waiting expression  
                   | ... ;

act expression = '#act', '(', name list, ')' ;

fin expression = '#fin', '(', name list, ')' ;

active expression = '#active', '(', name list, ')' ;

req expression = '#req', '(', name list, ')' ;

waiting expression = '#waiting', '(', name list, ')' ;

**Semantics:** History expressions can only be used in permission predicates (see section 14.1). History expressions may contain one or more of the following expressions:

- **#act** (*operation name*) . The number of times that *operation name* operation has been activated.
- **#fin** (*operation name*) . The number of times that the *operation name* operation has been completed.
- **#active** (*operation name*) . The number of *operation name* operations that are currently active.  
 Thus: **#active** (*operation name*) = **#act** (*operation name*) - **#fin** (*operation name*) .





- **#req** (*operation name*) . The number of requests that has been issued for the *operation name* operation.
- **#waiting** (*operation name*) . The number of outstanding requests for the *operation name* operation.  
Thus: **#waiting** (*operation name*) = **#req** (*operation name*) - **#act** (*operation name*) .

For all of these operators, the name list version **#history op**(*op1*, . . . , *opN*) is simply shorthand for **#history op**(*op1*) + . . . + **#history op**(*opN*).

**Examples:** Suppose at a point in the execution of a particular thread, three operations, A, B and C may be executed. A sequence of requests, activations and completions occur during this thread. This is shown graphically in figure 6.1.

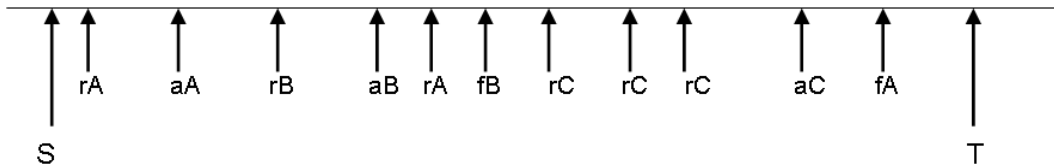


Figure 6.1: *History Expressions*

Here we use the notation *rA* to indicate a request for an execution of operation A, *aA* indicates an activation of A, *fA* indicates completion of an execution of operation A, and likewise for operations B and C. The respective history expressions have the following values after the interval [S,T]:

<b>#act</b> (A) = 1	<b>#act</b> (B) = 1	<b>#act</b> (C) = 1	<b>#act</b> (A,B,C) = 3
<b>#fin</b> (A) = 1	<b>#fin</b> (B) = 1	<b>#fin</b> (C) = 0	<b>#fin</b> (A,B,C) = 2
<b>#active</b> (A) = 0	<b>#active</b> (B) = 0	<b>#active</b> (C) = 1	<b>#active</b> (A,B,C) = 1
<b>#req</b> (A) = 2	<b>#req</b> (B) = 1	<b>#req</b> (C) = 3	<b>#req</b> (A,B,C) = 6
<b>#waiting</b> (A) = 1	<b>#waiting</b> (B) = 0	<b>#waiting</b> (C) = 2	<b>#waiting</b> (A,B,C) = 3

## 6.24 The Time Expression (VDM-RT)

**Syntax:** time expression = ‘time’;

**Semantics:** This is simply an easy way to refer to the current time on a given CPU. The time is provided as a natural number, with a resolution of 1 nsec.

**Examples:** If for example one would like to log when a certain operation takes place one can create an operation such as `logEnvToSys` below.



```
public logEnvToSys: nat ==> ()
logEnvToSys (pev) == e2s := e2s munion {pev |-> time};
```

## 6.25 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/class name where the construct is defined (see also section 13.1 and section 13.3.1 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 chapter 11) and in the post condition of specification statements (see section 12.16). 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 VDM-SL 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
```

For an example of the use of *old names*, consider the VDM++/VDM-RT instance variables defined as:

```
instance variables
  numbers: seq of nat := [];
  index   : nat := 1;
inv index not in set elems numbers;
```

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 chapter 11).

For a simple example of module/class names, suppose that a function called `build_rel` is defined (and exported) in a module/class 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/class we can access this function by in VDM-SL first importing the module `CGRel` then by using the following call

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

Note that in VDM++ and VDM-RT the `build_rel` function can additionally have an access modifier allowing access to it outside the defining class.

## 6.26 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 VDM interpreters 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
```

## 6.27 The Precondition Expression

**Syntax:** `expression = ...`  
                   | `precondition expression ;`

`precondition expression = 'pre_', '(', expression list, ')' ;`



**Semantics:** Assuming  $e$  is of function type the expression  $\text{pre}_-(e, e_1, \dots, e_n)$  is true if and only if the pre-condition of  $e$  is true for arguments  $e_1, \dots, e_m$  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  $\text{pre}_f(1, 0)$ ;
- true if  $h$  is bound to  $g$  since this equates to  $\text{pre}_g(1)$ ;
- true if  $h$  is bound to  $\text{lambda mk}_-(x, y) : \text{nat} * \text{nat} \ \& \ x \ \text{div} \ y$  since there is no pre-condition defined for this function.

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

When a function is defined to be total, that totality is with respect to the pre-condition. Consider an alternative definition of ‘ $f$ ’

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

The definition states the function is total for all values of type  $\text{nat} * \text{nat}$  where the second element of the pair is non-zero. Therefore, even though the function is not defined for values such as  $\text{mk}_-(4, 0)$  it is still a total function. On the other hand, the definition



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

is incorrect because it is not defined for values such as `mk_ (4, 0)` hence not total.



# Chapter 7

## 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 munion pattern
         | tuple pattern
         | record pattern
         | object 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 munion pattern = pattern, 'munion', pattern ;

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

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

object pattern = 'obj_', identifier, '(', [ field pattern list ], ')' 1;

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

field pattern = identifier, '|->', pattern ;

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 chapter). 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 VDM interpreters 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.

<sup>1</sup>**Note:** object pattern is only be used in VDM++ and VDM-RT. The object pattern is explained in 7.1



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 VDM interpreters 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.
12. An *object pattern* is explained in section 7.1

**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 VDM interpreters always ensure 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 sequence 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([l(i) | i in set inds l & l(i) < x])
    ^ [l(i) | i in set inds l & l(i) = x] ^
    quicksort([l(i) | i in set inds l & l(i) > 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. The following `map2seq` function converts a map to a seq of maplet:

```

public 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, l, -, -) in set elems GroupA }
```

The function `print_Expr` on page 58 also gives several examples of record patterns.

## 7.1 Object Pattern (VDM++ and VDM-RT)

**Syntax:** object pattern = `'obj_'`, identifier, `'('`, [ field pattern list ], `' ) '` ;

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

field pattern = identifier, `'|->'`, pattern ;

**Semantics:** An object pattern matches object references. An object is matched to the class identified by the class name led by the prefix `obj_`. If the class of the object is class identifier or any of its subclasses, separate instance variables named in an object pattern are matched against the object from left to right. It is not necessary to enumerate all the instance variables of a referenced class in an object pattern.

Object patterns may appear only where access to the instance variables named in the pattern would be permitted. So private instance variables can only be matched within operations of the same class. This also implies an object pattern may not appear in function definitions, preconditions, postconditions or invariants. The type checker will validate accessibility of each instance variable in the pattern. The value of referenced instance variables is not guaranteed to remain unchanged during matching.

**Examples:** Using an object pattern, an object can be matched by the values of its instance variables. The following simple example evaluates to `<STUDENT>` if and only if the object reference `person` points at an instance of the class `Student` or any of its subclasses, otherwise `<NOT_STUDENT>`:

```
cases person:
  obj_Student() -> <STUDENT>,
  others -> <NOT_STUDENT>
```



```
end
```

The above object pattern would match any instance of the `Student` class, or any subclass, for example an `Undergraduate` if such a subclass existed. The following example matches the instance variable `name` against `"John"`:

```
cases person:
  obj_Student(name |-> "John") -> <JOHN>,
  others -> <NOT_A_STUDENT_OR_NOT_JOHN>
end
```

An object pattern can also bind the value of instance variable. For example, the following expression binds `n` to the `name` instance variable of `person` if `person` is an instance of the class `Student`.

```
cases person:
  obj_Student(name |-> n) -> n,
  others -> ""
end
```

An object pattern may contain another object pattern. The following expression gives the name and department concatenated by `"@"` for `Students`, and add the title for `Professors`.

```
cases person:
  obj_Student
    (name |-> n, dept |-> obj_Department(name->dname))
    -> n^"@"^dname,
  obj_Professor(name |-> n) -> "Prof. "^n,
  others -> ""
end
```

An object pattern may appear in conjunction with other kinds of patterns. The following operation takes a sequence of `Professor` objects and returns titled names sorted into initials. The first pattern of the cases expression consists of a list concatenation pattern. The first element of the list is matched against an object pattern whose instance variable `name` is also matched against another sequence concatenation pattern of the initial character `initial` and the remaining:

```
professorDict: seq of Professor
==> map char to set of seq of char
```



```
professorDict(ps) == return cases ps:
  [obj_Professor(name |-> [initial]^remaining)]^rest ->
    let
      dict = professorDict(rest),
      name = "Prof. "^[initial]^remaining,
      names = if initial in set dom dict
               then dict(initial)
               else {}
    in
      dict ++ {initial |-> names union {name}},
      [-]^rest -> professorDict(rest), [] -> {|->}
end;
```

For example, the call `professorDict([new Professor("Smith"), new Professor("Scott"), new Professor("Adams")])` evaluates to:

```
{'A' |-> {"Prof. Adams"},
 'S' |-> {"Prof. Scott", "Prof. Smith"}}.
```

# Chapter 8

## Bindings

**Syntax:** `bind = set bind | seq bind | type bind ;`

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

`seq bind = pattern, 'in seq', expression ;`

`type bind = pattern, ':', type ;`

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

`multiple bind = multiple set bind  
                  | multiple seq bind  
                  | multiple type bind ;`

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

`multiple seq bind = pattern list, 'in seq', 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 *seq bind* the value is chosen from the sequence defined by the sequence 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, sequence or type. Notice that type binds **can** only be executed by the VDM interpreters in case the type can be deduced to be finite statically. This would require the VDM interpreters 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);
```

# Chapter 9

## Value (Constant) Definitions

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

**Syntax:** value definitions = **'values'**, [ access value definition,  
{ **';**', access value definition }, [ **';**' ] ] ;

access value definition = [ access ], value definition ;

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

**Semantics:** The value definition has the form:

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

where the `access` part only can be used in VDM++ and VDM-RT.

The global values (defined in a value definition) can be referenced at all levels in a VDM 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 VDM interpreters 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.

Details of the VDM++ and VDM-RT access specifiers can be found in section 13.3.3.

**Examples:** The example below, taken from [?] 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**

```
public p1: Period = mk_token("Monday day");  
private eid2 : ExpertId = mk_token(145);  
protected 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. The access modifiers **private**, **protected** and **public** can only be used in VDM++ and VDM-RT. A top-level VDM-SL specification can consist of specifications from a number of files or modules (see section 13.1). It is good practice not to let a value depend on values defined in other modules as the ordering is important.

# Chapter 10

## Declaration of Modifiable State Components

Syntactically the definition of state components that can be modified using VDM operations differ in VDM-SL compared with VDM++ and VDM-RT. Since VDM-SL is module based the state definition is similar to a monolithic record like construct. On the other hand VDM++ and VDM-RT are object-oriented and thus state components needs to be more flexible in order to enable inheritance of such definitions and thus they are defined in terms of instance variables. In the two sections in this chapter the two different ways of defining states is presented.

### 10.1 Instance Variables (VDM++ and VDM-RT)

Both an object instantiated from a class description and the class itself can have an internal state, also called the *instance variables* of the object or class. In the case of objects, we also refer to this state as the global state of the object.

**Syntax:** instance variable definitions = **'instance'**, **'variables'**,  
[ instance variable definition,  
{ **';**', instance variable definition } ] ;

instance variable definition = access assignment definition  
| invariant definition ;

access assignment definition = ([ access ], [ **'static'** ] | [ **'static'** ], [ access ]),  
assignment definition ;

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

invariant definition = **'inv'**, expression ;



**Semantics:** The section describing the internal state is preceded by the keyword **instance variables**. A list of instance variable definitions and/or invariant definitions follows. Each instance variable definition consists of an instance variable name with its corresponding type indication and may also include an initial value and access and **static** specifiers. Details of the access and **static** specifiers can be found in section 13.3.3.

It is possible to restrict the values of the instance variables by means of invariant definitions. Each invariant definition, involving one or more instance variables, may be defined over the values of the instance variables of objects of a class. All instance variables in the class including those inherited from superclasses are visible in the invariant expression. Each invariant definition must be a boolean expression that limits the values of the instance variables to those where the expression is true. All invariant expressions must be true during the entire lifetime of each object of the class.

The overall invariant expression of a class is all the invariant definitions of the class and its superclasses combined by logical **and** in the order that they are defined in 1) the superclasses and 2) the class itself.

**Example:** The following examples show instance variable definitions. The first class specifies one instance variable:

```
class GroupPhase

types

GroupName = <A> | <B> | <C> | <D> | <E> | <F> | <G> | <H>;
Team = ... -- as on page 25
Score::team : Team
    won      : nat
    drawn    : nat
    lost     : nat
    points   : nat;

instance variables
gps : map GroupName to set of Score;
inv forall gp in set rng gps &
    (card gp = 4 and
     forall sc in set gp & sc.won + sc.lost + sc.drawn <= 3)

end GroupPhase
```



## 10.2 The State Definition (VDM-SL)

If global variables are desired in a VDM-SL 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 VDM interpreters.

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

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 == invpred
init pat2 == initpred
end
```

A state identifier `idn` is declared of a specific type `typen`. The invariant `invpred` is a boolean expression denoting a property which must hold for the state `ident` at all times. `initpred` denotes a condition which must hold initially. It should be noticed that in order to use the VDM interpreters, 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 VDM interpreters to evaluate the `initpred` 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 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.

# Chapter 11

## Operation Definitions

Operations have already been mentioned in chapter 4. The general form is described here and for VDM++ and VDM-RT special operations called *constructors* which are used for constructing instances of a class are described in section 11.1. Note that the **async** keyword can only be used in the VDM-RT dialect.

**Syntax:** operation definitions = **'operations'**, [ access operation definition,  
{ **';**', access operation definition } , [ **' '** ] ] ;

access operation definition = { **'pure'**  
| **'async'**  
| access  
| **'static'** },  
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'  
                | 'is subclass responsibility' ;
```

```
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:** Operations in VDM are by default synchronous but if the keyword “**async**” is used in VDM-RT in front of an operation definition it means that that operation will be treated as an asynchronous operation. This means that the operation cannot have a return type and the thread calling an asynchronous operation will continue its own execution after having



requested the invocation of the asynchronous operation. Note that constructors cannot be declared asynchronous.

If an operation is declared “**pure**” it means that it is executed atomically when it is called from a functional context (from functions, invariants, pre or post-conditions). Otherwise calling a pure operation is no different to calling a normal operation, except that a pure operation has various constraints. The constraints are as follows:

- A pure operation cannot update state
- A pure operation cannot call an impure operation
- A pure operation cannot have permission predicates
- An operation overriding a pure operation must also be pure
- History counters cannot be used for pure operations
- A mutex cannot refer to a pure operation
- A pure operation must return a value
- A pure operation cannot be declared **async** since an asynchronous operation is required to have **void** as return type.
- The body of a thread cannot be a pure operation
- It is not allowed to call **exit** in a pure operation

In both VDM++ and VDM-RT the details of the access and **static** specifiers can be found in section 13.3.3. Note that a static operation may not call non-static operations, and self expressions cannot be used in the definition of a static operation.

The following example of an explicit operation updates the VDM-SL state `GroupPhase` and the VDM++ instance variables of class `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)
      elseif 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 chapter 12. The statement may access any state/instance 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)
                elseif sc.team = lt
                then mu(sc, lost |-> sc.lost + 1)
                else sc
            | sc in set gps(gp) } };
```

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

In VDM-SL the presence of such pre- and post-conditions the VDM interpreters will also create 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<sup>1</sup>:

```
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

<sup>1</sup>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.



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:

```
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
<b>pre</b> _Op1(1,mk_St(2))	a bound to 1 in state St with n bound to 2
<b>post</b> _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
<b>post</b> _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
<b>post</b> _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
<b>pre</b> _Op1(1)	a bound to 1
<b>post</b> _Op1(1,2)	a bound to 1, b bound to 2
<b>post</b> _Op2(2)	b bound to 2
<b>post</b> _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  $\text{COND}_1, \dots, \text{COND}_n$  are identifiers that describe the kind of error which can be raised<sup>2</sup>. The condition expressions  $c_1, \dots, c_n$  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  $r_1, \dots, r_n$  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.

An operation definition making use of an **errs** clause essentially gets an effective pre-condition which is a disjunction of the original pre-condition and all the condition expressions  $c_1, \dots, c_n$ . The effective post-condition in these cases becomes a disjunction of the conjuncts ( $\text{orig\_pre}$  **and**  $\text{orig\_post}$ ) and the  $c_1$  **and**  $r_1, \dots, c_n$  **and**  $r_n$ .

Superficially there appears to be some redundancy between exceptions and pre-conditions here. However there is a conceptual distinction between them that 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 VDM-SL example of an operation uses the following state definition:

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

The next VDM++/VDM-RT example of an operation uses the following instance variable definition:

```
instance variables
  q : Queue
```

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
```

<sup>2</sup>Notice that these names are purely of mnemonic value, i.e. semantically they are not important.



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 VDM interpreters for VDM-SL models create a function here:

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

## 11.1 Constructors (VDM++ and VDM-RT)

Constructors are operations that have the same name as the class in which they are defined and which create new instances of that class. Their return type must therefore be the same class name, and if a return value is specified this should be **self** though this can optionally be omitted. Finally, since a constructor is used for initialising a new instance of a class, it is not permitted to declare a constructor **static**.

Multiple constructors can be defined in a single class using operation overloading as described in section 13.3.1.

# Chapter 12

## Statements

In this chapter 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.

### 12.1 Let Statements

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

let statement = ‘**let**’, local definition, { ‘,’, local definition },  
                          ‘**in**’, statement ;

let be statement = ‘**let**’, multiple 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 chapter 5.

**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  $p_1, \dots, p_n$  are patterns,  $e_1, \dots, e_n$  are expressions which match the corresponding patterns  $p_i$ , and  $s$  is a statement, of any type, involving the pattern identifiers of  $p_1, \dots, p_n$ . It denotes the evaluation of the statement  $s$  in the context in which the patterns  $p_1, \dots, p_n$  are matched against the corresponding expressions  $e_1, \dots, e_n$ .

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.

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

```
let mb be st e in s
```

where  $mb$  is a multi-binding of one or more patterns (mostly just one pattern) to a set value (or a type),  $e$  is a boolean expression, and  $s$  is a statement, involving the pattern identifiers of the patterns from  $mb$ . The **be st**  $e$  part is optional. The expression denotes the evaluation of the statement  $s$  in the context where all the patterns from  $mb$  has been matched against an element in the set (or type) from  $mb$ <sup>1</sup>. 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` from the class `GroupPhase` 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) ==
```

<sup>1</sup>Remember that only the set and sequence bindings can be executed by means of the VDM interpreters.



```

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 12.2) here; this is used rather than a **let** statement since the right-hand side is an operation call, and therefore is not an expression.

## 12.2 The Define Statement

**Syntax:**    statement = ...  
                   |    def statement  
                   |    ... ;

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*<sup>2</sup>.

<sup>2</sup>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 7.



**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`, in VDM++ from class `GroupPhase` 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}])
```

## 12.3 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 imperative 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 a VDM-SL state definition

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

or in the context of a VDM++ instance variables

```
instance variables
  x: nat;
  y: nat;
  l: seq1 of nat;
```

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
  )
```

## 12.4 The Assignment Statement

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

general assign statement = assign statement  
                               | multiple 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 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 in the order given in the statement, and normal invariant processing and thread switching and statement durations can occur. But once all of the right-hand values have been



obtained, they are assigned to the left-hand variables in one atomic step, which occurs without invariant checking, thread switching or extra duration. Given the types  $T_1, \dots, T_N$  of the respective state designators  $sd_1, \dots, sd_N$  it is as if the **atomic** statement is evaluated as follows:

```
let t1 : T1 = ec1,
    ...
    tN : TN = ecN in
(
  -- turn off invariants, threading and durations
  sd1 := t1;
  ...
  sdN := tN;
  -- turn on invariants, threading and durations
  -- and check that invariants hold.
);
```

**Examples:** The operation in the previous example (Swap) illustrated normal assignment. The operation `Win_sd`, a refinement of `Win` on page 106 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)
                  elseif 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 54. It demonstrates the use of state designators to modify the contents of a specific sequence index, using the VDM-SL state `St` or the VDM++ instance variables defined on page 115.

**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)
  );
```

The following VDM++ example illustrates multiple assignment.

**class C****instance variables**

```
size : nat;
l : seq of nat;
inv size = len l
```

**operations**

```
add1 : nat ==> ()
add1 (x) ==
  ( l := [x] ^ l;
    size := size + 1 );

add2 : nat ==> ()
add2 (x) ==
```



```

atomic (l := [x] ^ l;
        size := size + 1)

end C

```

Here, in `add1` the invariant on the class's instance variables is broken, whereas in `add2` using the multiple assignment, the invariant is preserved.

## 12.5 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 6.4 except for the alternatives which are statements (and that the **else** part is optional)<sup>3</sup>.

The semantics for the *cases statement* corresponds to the *cases expression* described in section 6.4 except for the alternatives which are statements.

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

<sup>3</sup>If the **else** part is omitted semantically it is like using **else skip**.





```
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 60):

```
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)
  elseif 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) \ {sc} &
                (sc.points > sc'.points) or
                (sc.points = sc'.points and
                 sc.won > sc'.won)).team)
  -- 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

```

## 12.6 For-Loop Statements

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

sequence for loop = 'for', pattern bind, 'in', 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 i.e. 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  $l$ .

**Examples:** The operation `Remove` demonstrates the use of a *sequence-for* loop to remove all occurrences 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
  )
```

## 12.7 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  
);
```

## 12.8 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 VDM interpreters will use an underdetermined<sup>4</sup> semantics even though this construct is called a non-deterministic statement.

**Examples:** Using the VDM-SL state definition

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

or the VDM++ instance variables

```
instance variables
```

<sup>4</sup>Even though the user of the VDM interpreters 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.



```
x:nat;
y:nat;
l:seq1 of nat;
```

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  $l(x, \dots, y)$  to the head of the subsequence and `BubbleMax` “bubbles” the maximum value in the subsequence  $l(x, \dots, y)$  to the last index in the subsequence. `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:\keyw{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));

```

## 12.9 The Call Statement

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

For VDM-SL call statements are defined as:

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

For VDM++ and VDM-RT call statements are defined as:

call statement = [ object designator, ‘.’ ], name,  
                   ‘(’, [ expression list ], ‘)’ ;

object designator = name  
                   | self expression  
                   | new expression  
                   | object field reference  
                   | object apply ;

object field reference = object designator, ‘.’, identifier ;

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

**Semantics:** In VDM-SL the *call statement* has the form:

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



In VDM++ and VDM-RT the *call statement* can additionally have the form:

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

The *call statement* calls an operation, *opname*, (in a VDM++ and VDM-RT context it can also be on a specific object, *object*), 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 call do.

In VDM++ and VDM-RT if an *object designator* is specified it must yield an object reference to an object of a class in which the operation *opname* is defined, and then the operation must be specified as public. If no *object designator* is specified the operation will be called in the current object. If the operation is defined in a superclass, it must have been defined as public or protected.

**Examples:** In VDM-SL 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.

In VDM++ and VDM-RT this `Stack` example can be made like:

```
class Stack

instance variables
  stack: seq of Elem := [];
```



**operations**

```
public Reset: () ==> ()  
Reset() ==  
    stack := [];  
  
public Pop: () ==> Elem  
Pop() ==  
    def res = hd stack in  
        (stack := tl stack;  
         return res)  
pre stack <> []  
post stack~ = [RESULT] ^ stack  
  
end Stack
```

In the example the operation `Reset` does not have any parameters and does not return a value whereas the operation `Pop` returns the top element of the stack. The stack could be used as follows:

```
( dcl stack := new Stack();  
  stack.Reset();  
  ....  
  top := stack.Pop();  
)
```

Inside class `Stack` the operations can be called as shown below:

```
Reset();  
....  
top := Pop();
```

Or using the **self** reference:

```
self.Reset();  
top := self.Pop();
```



## 12.10 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()
```

For instance in VDM++, we can extend the *stack* class from the previous section with an operation which examines the top of the stack:

```
public Top : () ==> Elem  
Top() ==  
  return (hd stack)  
pre stack <> [];
```

## 12.11 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`.

## 12.12 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 VDM interpreters 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 124 does not exclude the possibility that the number to be square rooted might be negative. If we do not wish this to be a pre-condition we can 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
  )

```

## 12.13 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 12.6 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.

## 12.14 Start and Start List Statements (VDM++ and VDM-RT)

**Syntax:** statement = ...  
                   | start statement  
                   | start list statement ;

start statement = '**start**', '(', expression, ')';

start list statement = '**startlist**', '(', expression, ')';

**Semantics:** The *start* and *start list* statements have the form:

```

start (aRef)
startlist (aRef_s)

```

If a class description includes a thread (see chapter 15), each object created from this class will have the ability to operate as a stand-alone virtual machine, or in other terms: the object has its own processing capability. In this situation, a *new expression* creates the 'process' leaving it in a waiting state. For such objects VDM++ and VDM-RT has a mechanism to change the waiting state into an active state<sup>5</sup> in terms of a predefined operation, which can be invoked through a *start statement*.

The explicit separation of object creation and start provides the possibility to complete the initialisation of a (concurrent) system *before* the objects start exhibiting their described behaviour, in this way avoiding problems that may arise when objects are referred to that are not yet created and/or connected.

<sup>5</sup>When an object is in an active state, its behaviour can be described using a thread (see chapter 15).



A syntactic variant of the start statement is available to start up a number of active objects in arbitrary order: the *start list statement*. The parameter `aRef_s` to **startlist** must be a set of object references to objects instantiated from classes containing a thread.

**Examples:** Consider the specification of an operating system. A component of this would be the daemons and other processes started up during the boot sequence. From this perspective, the following definitions are relevant:

#### types

```
runLevel = nat;
```

```
Process = Kerneld | Ftpd | Syslogd | Lpd | Httpd
```

#### instance variables

```
pInit : map runLevel to set of Process
```

where `Kerneld` is an object reference type specified elsewhere, and similarly for the other processes listed.

We can then model the boot sequence as an operation:

```
bootSequence : runLevel ==> ()
bootSequence(rl) ==
  for all p in set pInit(rl) do
    start(p);
```

Alternatively we could use the **startlist** statement here:

```
bootSequenceList : runLevel ==> ()
bootSequenceList(rl) ==
  startlist(pInit(rl))
```

## 12.15 Stop and Stop List Statements (VDM++ and VDM-RT)

**Syntax:** statement = ...  
                   | stop statement  
                   | stop list statement ;

stop statement = 'stop', '(', expression, ')';





stop list statement = **'stoplist'**, **'('**, expression, **)'** ;

**Semantics:** The *stop* and *stop list* statements have the following form:

```
stop (aRef)
stoplist (aRef_s)
```

The stop statement can be used to terminate a thread or to prevent a periodic thread from being invoked again, whereby a thread is identified by the object reference *aRef*. The stop list statement is available to stop a number of active objects, in arbitrary order. The parameter *aRef\_s* to **stoplist** must be a set of object references. A few additional remarks can be made:

1. **'self'** can be passed as an argument to *stop* or *stoplist*, effectively killing the current thread
2. Note that *thread identifiers* can *not* be passed as an argument
3. in contrast to *start* and *startlist*, *stop* and *stoplist* will block until the thread(s), identified by their object identifiers that are passed as an argument, are dead; if **'self'** is part of the set, it will be stopped last
4. passing the object identifier of a task that has never been started or is already dead, will yield a run-time error
5. objects can be restarted by calling *start* and *startlist* again, which will create a fresh and unique *thread identifier* for each thread.
6. it is only possible to stop threads that are running on the same CPU

## 12.16 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 chapter 11). 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.)

## 12.17 The Duration Statement (VDM-RT)

**Syntax:** statement = ...  
| duration statement ;

duration statement = 'duration', '(', expression, ')', statement ;

**Semantics:** The duration statement is a runtime directive to the VDM interpreters telling it that when incrementing the internal clock for the enclosed statement, the value (an expression that must yield a natural number as its return type, otherwise causing a run-time error) given in the duration statement should be used instead of the increment which would normally be computed for that statement. Thus the duration statement provides a mechanism to override the VDM interpreter's default execution time computation. Note that the execution of the expression is done in zero time and cannot cause thread switches.

**Example:** First a simple example:

```
while n < 10 do
  duration(10) n := n + 1;
```

In this example, assuming that this loop is not executed in the context of an enclosing duration statement, on each iteration of the loop the VDM interpreters will increment its internal



clock by 10 time units (nanoseconds), rather than computing the amount of time required to execute the statement  $n := n + 1$ .

If duration statements are nested, the outermost one takes precedence and the remainder are ignored. For instance

```
duration(30)
( n := 1;
  while n < 10 do
    duration(10) n := n + 1;
  )
```

The outer duration statement takes precedence, so assuming this is not executed in the context of an enclosing duration statement, the VDM interpreters would increment its internal clock by 30 time units when executing this statement.

Note that nesting can occur due to operation calls. Consider the following example:

```
op1 : nat ==> nat
op1(m) ==
  duration (20) return m + 1;

op2 : () ==> nat
op2() ==
( dcl n : nat := 3;
  duration(10) n := op1(1);
  return n)
```

When executing `op2`, if the call to `op1` is executed, the duration statement in `op1` will be overridden by the duration statement in the environment of the call. Thus in `op2` following execution of the statement  $n := op1(1)$ ; the internal clock is incremented by 10 time units only.

```
( n := 1; while n < 10 do duration(n) n := n + 1 )
```

The final example demonstrates the use of a general expression as the argument to the duration statement, whereby the consecutive executions of the body of the while loop take more time as  $n$  increases.



## 12.18 The Cycles Statement (VDM-RT)

**Syntax:** `statement = ...`  
                   `| cycles statement ;`

`cycles statement = 'cycles', '(', expression, ')', statement ;`

**Semantics:** The cycles statement is a runtime directive to the VDM interpreters telling it that when incrementing the internal clock for the enclosed statement, the value (an expression that must yield a natural number as its return type, otherwise causing a run-time error) given in the cycles statement should be used as an indication of how many clock cycles that the enclosed statement should be incremented by instead of the increment which would normally be computed for that statement. Thus the cycles statement provides a mechanism to override the VDM interpreter's default execution time computation similar to the duration statement but in a way that is relative to the speed of the CPU that the computation is carried out on. Note that the execution of the expression is done in zero time and cannot cause thread switches.

**Example:** First a simple example:

```
while n < 10 do
  cycles(1000) n := n + 1;
```

In this example, assuming that this loop is not executed in the context of an enclosing cycles statement, on each iteration of the loop the VDM interpreters will increment its internal clock by the time it will take to process 1000 instructions on the given CPU (relative to its capacity), rather than computing the amount of time required to execute the statement `n := n + 1`.

If cycles statements are nested, the outermost one takes precedence and the remainder are ignored. For instance

```
cycles(3000) (
  n := 1;
  while n < 10 do
    cycles(1000) n := n + 1;
)
```

The outer cycles statement takes precedence, so assuming this is not executed in the context of an enclosing cycles statement, the interpreter would increment its internal clock by the time it takes to process 3000 instructions on the given CPU when executing this statement.

Note that nesting can occur due to operation calls. Consider the following example:



```
op1 : nat ==> nat
op1(m) ==
    cycles (2000) return m + 1;

op2 : () ==> nat
op2() ==
    (dcl n : nat := 3;
     cycles(1000)  n := op1(1);
     return n)
```

When executing `op2`, if the call to `op1` is executed, the `cycles` statement in `op1` will be overridden by the `cycles` statement in the environment of the call. Thus in `op2` following execution of the statement `n := op1(1);` the internal clock is incremented by the time it takes to process 1000 instructions on the given CPU only.

```
( n := 1; while n < 10 do cycles(n) n := n + 1 )
```

The final example demonstrates the use of a general expression as the argument to the `cycles` statement, whereby the consecutive executions of the body of the while loop take more time as  $n$  increases.

# Chapter 13

## Top-level Specification in VDM

The top-level specification structure differs significantly between the VDM-SL approach and the VDM++ and VDM-RT approach. In VDM-SL the ISO standard prescribes a flat-language but here a modular extension is also enabled using imports and exports primitives. In VDM++ and VDM-RT structuring is done using object-oriented classes that can inherit constructs between them controlled by access modifiers. These two different approaches are explained in the two sections in this chapter.

### 13.1 Top-level Specification in VDM-SL

In the previous chapters 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 (note that in VDM-10 it is possible to have access to standard modules also from a flat VDM-SL specification).

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

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

#### 13.1.1 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
                  | traces 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
  list:seq1 of nat
end

functions

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

operations

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

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



### 13.1.2 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.

#### 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 two 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 any 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 } ;

Note that the identifier after **module** and **end** should be identical.





## 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  
| export values signature  
| export functions signature  
| export operations signature ;

export types signature = **'types'**, type export,  
| { ';' , type export }, [ ';' ] ;

type export = [ **'struct'** ], name ;

export values signature = **'values'**, value signature,  
| { ';' , value signature }, [ ';' ] ;

value signature = name list, ':', type ;

export functions signature = **'functions'**, function signature,  
| { ';' , function signature }, [ ';' ] ;

function signature = name list, [ type variable list ], ':',  
function type ;

export 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 the **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 chapter 5) 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;
                    deposit: account * real -> account;
                    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.

### 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.

A module that imports a type may only access the internal structure of that type if it was exported with the **struct** keyword; this includes invoking the type constructor and using the selector functions. If the type was exported without the **struct** keyword then the importing module may refer to the type, and call functions from the imported module that create elements of the type, but it cannot invoke the type constructor, or use the selector functions.

When a composite type is exported with **struct** then its constructor is referred to, in the importing module, by **mk\_ExportingMod `RecordName**. If the composite type is renamed on import, to `ImportedRecName` say, then the constructor is referred to by **mk\_ImportedRecName**.

**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,-,-)) ==
    let m = atmc_digval(m1) * 10 + atmc_digval(m2)
    in m > 0 and m <= 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`. This notwithstanding, all values of type `digit` manipulated in `ATMCard` must satisfy the invariant.

## 13.2 Top-level Specification in VDM++ and VDM-RT

In the previous chapters VDM constructs such as types, expressions, statements, functions and operations have been described. A number of these constructs can constitute the definitions inside a class definition. A top-level specification, or document, is composed of one or more class definitions. Note that only in VDM-RT it is possible to have a **system** class.

**Syntax:** document = ( class | system ), { class | system } ;

## 13.3 System (VDM-RT)

In order to be able to describe distributed systems VDM-RT includes a notion of a system that describes how different parts of the system modelled are deployed to different Core Processing Units (CPUs) and communication busses connecting the CPUs together. Syntactically the system is described exactly like ordinary classes described below in Section 13.3.1, except that the keyword “**system**” instead of the keyword “**class**” is used.

**Syntax:** system = ‘**system**’, identifier,  
[ class body ],  
‘**end**’, identifier ;

class body = definition block, { definition block } ;

definition block = type definitions  
| value definitions  
| function definitions  
| operation definitions  
| instance variable definitions



```

|   synchronization definitions
|   thread definitions ;

```

**Semantics:** Each system description has the following parts:

- A system header with the system name.
- An optional *system body*.
- A system tail.

The system name as given in the system header is the defining occurrence of the name of the class. A system name is globally visible, i.e. visible in all other classes/systems in the specification.

The system name in the class header must be the same as the system name in the system tail. Furthermore, defining system names must be unique throughout the specification.

The special thing about the system is that it can make use of special implicitly defined classes called CPU and BUS. It is not possible to create instances of the system, but instances made of CPU and BUS will be created at initialisation time. Note that CPU and BUS cannot be used outside the system definition.

The instances of CPU and BUS must be made as instance variables and the definition must use constructors. The constructor for the CPU class takes two parameters: the first one indicates the primary scheduling policy used for the CPU whereas the second parameter provides the capacity of the CPU (indicated as number of instructions Per Second or Hz – NB. the (time unit) step size of time is 1 nanosecond). The constructor for the BUS class takes three parameters. The first one indicates the kind of bus, the second one the capacity of the bus (its band width in bytes per second) and finally the third parameter gives a set of CPU instances connected together by the given BUS instance.

The currently supported primary scheduling policies for the CPU are:

**<FP>:** Fixed Priority

**<FCFS>:** First Come First Served

The currently supported primary scheduling policy for the BUS is:

**<FCFS>:** First Come First Served

The CPU class has member operations called `deploy` and `setPriority`. The `deploy` operation takes one significant parameter which must be an object that is declared as a static instance variable inside the system<sup>1</sup>. The semantics of the `deploy` operation is that execution of all functionality inside this object will take place on the CPU that it has been deployed to. The `setPriority` operation takes two parameters where the first must be the name

<sup>1</sup>It is also allowed to take a string as a second parameter for future extensions but that is ignored at the moment.





of a public operation that has been deployed to the CPU and the second parameter is a natural number. The semantics of the `setPriority` operation is that the given operation is assigned the given priority (the second parameter). This will be used when fixed priority scheduling is used on the given CPU. Per default operations that are not explicitly assigned a priority using the `setPriority` operation are assigned a default priority of 1.

The system “class” is limited in the way that it can only contain:

**Instance variables:** The only instances that can be declared in the system “class” is of the special classes CPU and BUS as well as static instances of the different system components that one wishes to allocate to different CPU’s.

**Constructor:** The actual deployment of instances to CPU’s and setting of priorities for the different operations is set inside the constructor which is the only operation that can be placed in the system “class”. The only kind of statement that can be used inside this constructor is a block statement with a sequence of invocations of the special `deploy` and `setPriority` operations.

In addition there are limitations with respect to the use of static declarations for instances that are deployed to different CPU’s. Basically the user should ensure that only one instance is deployed to a CPU if the class the instance comes from contains any static operations or functions. In case a static instance variable is used it is accessed directly (without any communication over the busses), so this in essence not proper from a distribution standpoint. Thus, all instance variables of instances to be deployed should only be accessed through the use of operations.

**Example:** The system class could for example be defined as:

```
system Simple

instance variables
  static public a : A := new A();
  static public b : B := new B();
  -- define the first CPU with fixed priority scheduling
  -- and 22E6 Hz
  CPU1 : CPU := new CPU (<FP>, 22E6);

  static public c : C := new C();
  -- define the second CPU with fixed priority scheduling
  -- and 11E6 Hz
  CPU2 : CPU := new CPU (<FP>, 11E6);

  -- create a communication bus that links the three
  -- CPU's together
  BUS1 : BUS := new BUS (<CSMACD>, 72E3, {CPU1, CPU2})
```



```

operations
  public Simple: () ==> Simple
  Simple () ==
    ( -- deploy a on CPU1
      CPU1.deploy(a);
      -- deploy b on CPU1
      CPU1.deploy(b);
      -- deploy c on CPU2
      CPU2.deploy(c, "CT");
      -- "CT" is a label here which is ignored
    );
end Simple

```

where A, B and C all are defined as classes.

### 13.3.1 Classes

Compared to the standard VDM-SL language, VDM++ and VDM-RT have been extended with classes. In this section, the use of classes to create and structure a top-level specification will be described. With the object oriented facilities offered by VDM++ and VDM-RT it is possible to:

- Define classes and create objects.
- Define associations and create links between objects.
- Make generalisation and specialisation through inheritance.
- Describe the functional behaviour of the objects using functions and operations.
- Describe the dynamic behaviour of the system through threads and synchronisation constraints.

Before the actual facilities are described, the general layout of a class is described.

**Syntax:** class = **'class'**, identifier, [ inheritance clause ],  
                   [ class body ],  
                   **'end'**, identifier ;

inheritance clause = **'is subclass of'**, identifier, { **' , '**, identifier } ;

class body = definition block, { definition block } ;



```
definition block = type definitions
                  | value definitions
                  | function definitions
                  | operation definitions
                  | instance variable definitions
                  | synchronization definitions
                  | thread definitions
                  | traces definitions ;
```

**Semantics:** Each class description has the following parts:

- A class header with the class name and an optional *inheritance clause*.
- An optional *class body*.
- A class tail.

The class name as given in the class header is the defining occurrence of the name of the class. A class name is globally visible, i.e. visible in all other classes in the specification.

The class name in the class header must be the same as the class name in the class tail. Furthermore, defining class names must be unique throughout the specification.

The (optional) class body may consist of:

- A set of *value definitions* (constants).
- A set of *type definitions*.
- A set of *function definitions*.
- A set of *instance variable definitions* describing the internal state of an object instantiated from the class. State invariant expressions are encouraged but are not mandatory.
- A set of *operation definitions* that can act on the internal state.
- A set of the *synchronization definitions*, specified either in terms of permission predicates or using mutex constraints.
- A set of *thread definitions* that describe the thread of control for active objects.
- A set of *traces* that are used to indicate the sequences of operation calls for which test cases are desired to be produced automatically.

In general, all constructs defined within a class must have a unique name, e.g. it is not allowed to define an operation and a type with the same name. However, it is possible to *overload* function and operation names (i.e. it is possible to have two or more functions with the same name and two or more operations with the same name) subject to the restriction that the types of their input parameters should not overlap. That is, it should be possible using static type checking alone to determine uniquely and unambiguously which function/operation definition corresponds to each function/operation call. Note that this applies not only to



functions and operations defined in the local interface of a class but also to those inherited from superclasses. Thus, for example, in a design involving multiple inheritance a class C may inherit a function from a class A and a function with the same name from a class B and all calls involving this function name must be resolvable in class C.

### 13.3.2 Inheritance

The concept of inheritance is essential to object orientation. When one defines a class as a subclass of an already existing class, the definition of the subclass introduces an extended class, which is composed of the definitions of the superclass together with the definitions of the newly defined subclass.

Through inheritance, a subclass inherits from the superclass:

- Its instance variables. This also includes all invariants and their restrictions on the allowed modifications of the state.
- Its operation and function definitions.
- Its value and type definitions.
- Its synchronization definitions as described in section 14.2.
- Its thread definitions as described in chapter 15.

A name conflict occurs when two constructs of the same kind and with the same name are inherited from different superclasses. Name conflicts must be explicitly resolved through *name qualification*, i.e. prefixing the construct with the name of the superclass and a ```-sign (back-quote).

**Example:** In the first example, we see that inheritance can be exploited to allow a class definition to be used as an abstract interface which subclasses must implement:

```
class Sort

  instance variables
    protected data : seq of int

  operations

    initial_data : seq of int ==> ()
    initial_data (list) ==
      data := list;

    sort_ascending : () ==> ()
    sort_ascending () == is subclass responsibility;
```



```
end Sort

class SelectionSort is subclass of Sort

  functions

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

  operations

    sort_ascending : () ==> ()
    sort_ascending () == selectSort(1);

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

            temp := data(mi);
            data(mi) := data(i);
            data(i) := temp;
            selectSort(i+1)
           )

end SelectionSort
```

Here the class `Sort` defines an abstract interface to be implemented by different sorting algorithms. One implementation is provided by the `SelectionSort` class.

The next example clarifies how name space clashes are resolved.

```
class A
```



```

instance variables
  i: int := 1;
  j: int := 2;
end A

class B is subclass of A
end B

class C is subclass of A
instance variables
  i: int := 3;
end C

class D is subclass of B,C
operations
  GetValues: () ==> seq of int
  GetValues() ==
    return [
      A`i, -- equal to 1
      B`i, -- equal to 1 (A`i)
      C`i, -- equal to 3
      j    -- equal to 2 (A`j)
    ]
end D

```

In the example objects of class D have 3 instance variables: A`i, A`j and C`j. Note that objects of class D will have only one copy of the instance variables defined in class A even though this class is a common super class of both class B and C. Thus, in class D the names B`j, C`j, D`j and j are all referring to the same variable, A`j. It should also be noticed that the variable name i is ambiguous in class D as it refers to different variables in class B and class C.

### 13.3.3 Interface and Availability of Class Members

In VDM++ and VDM-RT definitions inside a class are distinguished between:

**Class attribute:** an attribute of a class for which there exists exactly one incarnation no matter how many instances (possibly zero) of the class may eventually be created. Class attributes in VDM++ and VDM-RT correspond to **static** class members in languages like C++ and Java. Class (static) attributes can be referenced by prefixing the name of the attribute with the name of the class followed by a `-sign (back-quote), so that, for example, `ClassName`val` refers to the value `val` defined in class `ClassName`.



**Instance attribute:** an attribute for which there exists one incarnation for each instance of the class. Thus, an instance attribute is only available in an object and each object has its own copy of its instance attributes. Instance (non-static) attributes can be referenced by prefixing the name of the attribute with the name of the object followed by a dot, so that, for example, `object.op()` invokes the operation `op` in the object denoted by `object` (provided that `op` is visible to `object`).

Functions, operations, instance variables and constants<sup>2</sup> in a class may be either class attributes or instance attributes. This is indicated by the keyword **static**: if the declaration is preceded by the keyword **static** then it represents a class attribute, otherwise it denotes an instance attribute.

Other class components are by default always either class attributes or instance attributes as follows:

- Type definitions are always class attributes.
- Thread definitions are always instance attributes. Thus, each active object has its own thread(s).
- Synchronization definitions are always instance attributes. Thus, each object has its own “history” when it has been created.

In addition, the interface or accessibility of a class member may be explicitly defined using an access specifier: one of **public**, **private** or **protected**. The meaning of these specifiers is:

**public:** Any class may use such members

**protected:** Only subclasses of the current class may use such members

**private:** No other class may use such members - they may only be used in the class in which they are specified.

The default access to any class member is **private**. That is, if no access specifier is given for a member it is private.

This is summarized in table 13.1. A few provisos apply here:

- Granting access to instance variables (i.e. through a **public** or **protected** access specifier) gives both read and write access to these instance variables.
- Public instance variables may be read (but not written) using the dot (for object instance variables) or back-quote (for class instance variables) notation e.g. a public instance variable `v` of an object `o` may be accessed as `o.v`.
- Access specifiers may only be used with type, value, function, operation and instance variable definitions; they cannot be used with thread or synchronization definitions.

<sup>2</sup>In practice, constants will generally be static – a non-static constant would represent a constant whose value may vary from one instance of the class to another which would be more naturally represented by an instance variable.



	<b>public</b>	<b>protected</b>	<b>private</b>
Within the class	✓	✓	✓
In a subclass	✓	✓	×
In an arbitrary external class	✓	×	×

Table 13.1: Summary of Access Specifier Semantics

- It is not possible to convert a class attribute into an instance attribute, or vice-versa.
- For inherited classes, the interface to the subclass is the same as the interface to its super-classes extended with the new definitions within the subclass.
- Access to an inherited member cannot be made more restrictive e.g. a public instance variable in a superclass cannot be redeclared as a private instance variable in a subclass.

**Example** In the example below use of the different access specifiers is demonstrated, as well as the default access to class members. Explanation is given in the comments within the definitions.

```

class A

types
public Atype = <A> | <B> | <C>

values
public Avalue = 10;

functions
public compare : nat -> Atype
compare(x) ==
  if x < Avalue
  then <A>
  elseif x = Avalue
  then <B>
  else <C>

instance variables
public v1: nat;
private v2: bool := false;
protected v3: real := 3.14;

operations

```





```
protected AInit : nat * bool * real ==> ()
AInit(n,b,r) ==
  (v1 := n;
   v2 := b;
   v3 := r)
end A

class B is subclass of A

instance variables
v4 : Atype --inherited from A

operations

BInit: () ==> ()
BInit() ==
  (AInit(1,true,2.718); --OK: can access protected members
                        --in superclass
   v4 := compare(v1);   --OK since v1 is public
   v3 := 3.5;           --OK since v3 protected and this
                        --is a subclass of A
   v2 := false         --illegal since v2 is private to A
  )

end B

class C

instance variables
a: A := new A();
b: B := new B();

operations

CInit: () ==> A`Atype--types are class attributes
CInit() ==
  (a.AInit(3,false,1.1);
                        --illegal since AInit is protected
   b.BInit();          --illegal since BInit is (by default)
                        --private
   let - = a.compare(b.v3) in skip;
                        --illegal since C is not subclass
```



```
                                --of A so b.v3 is not available
return b.compare(B`Avalue)
                                --OK since compare is a public instance
                                --attribute and Avalue is public class
                                --attribute in B
)
end C
```



# Chapter 14

## Synchronization Constraints (VDM++ and VDM-RT)

In general a complete system contains objects of a passive nature (which only react when their operations are invoked) and active objects which ‘breath life’ into the system. These active objects behave like virtual machines with their own processing thread of control and after start up they do not need interaction with other objects to continue their activities. In another terminology a system could be described as consisting of a number of active clients requesting services of passive or active servers. In such a parallel environment the server objects need synchronization control to be able to guarantee internal consistency, to be able to maintain their state invariants. Therefore, in a parallel world, a passive object needs to behave like a Hoare monitor with its operations as entries.

If a sequential system is specified (in which only one thread of control is active at a time) only a special case of the general properties is used and no extra syntax is needed. However, in the course of development from specification to implementation more differences are likely to appear.

The following default synchronization rules for each object apply in VDM++ and VDM-RT:

- operations are to be viewed as though they are atomic, from the point of the caller;
- operations which have no corresponding permission predicate are subject to no restrictions at all;
- synchronization constraints apply equally to calls within an object (i.e. one operation within an object calls another operation within that object) and outside an object (i.e. an operation from one object calls an operation in another object);
- operation invocations have the semantics of a rendez-vous (as in Ada, see [?]) in case two active objects are involved. Thus if an object  $O_1$  calls an operation  $o$  in object  $O_2$ , if  $O_2$  is currently unable to start operation  $o$  then  $O_1$  blocks until the operation may be executed. Thus invocation occurs when both the calling object and the called object are ready. (Note here a slight difference from the semantics of Ada: in Ada both parties to the rendez-vous are active objects; in VDM++ and VDM-RT only the calling party is active).



The synchronization definition blocks of the class description provide the user with ways to override the defaults described above.

**Syntax:** synchronization definitions = **'sync'**, [ synchronization ] ;

synchronization = permission predicates ;

**Semantics:** Synchronization is specified in VDM++ and VDM-RT using permission predicates.

## 14.1 Permission Predicates

The following gives the syntax used to state rules for accepting the execution of concurrently callable operations. Some notes are given explaining these features.

**Syntax:** permission predicates = permission predicate, { **';**',  
permission predicate } ;

permission predicate = **'per'**, name, **'=>'**, expression  
| mutex predicate ;

mutex predicate = **'mutex'**, **'('**, (**'all'** | name list, **' '**) ;

**Semantics:** Permission to accept execution of a requested operation depends on a guard condition in a (deontic) permission predicate of the form:

**per** operation name => guard condition

The use of implication to express the permission means that truth of the guard condition (expression) is a necessary but not sufficient condition for the invocation. The permission predicate is to be read as stating that if the guard condition is false then there is non-permission. Expressing the permission in this way allows further similar constraints to be added without risk of contradiction through inheritance for the subclasses. There is a default for all operations:

**per** operation name => **true**

but when a permission predicate for an operation is specified this default is overridden.

Guard conditions can be conceptually divided into:

- a *history guard* defining the dependence on events in the past;
- an *object state guard*, which depends on the instance variables of the object, and
- a *queue condition guard*, which depends on the states of the queues formed by operation invocations (messages) awaiting service by the object.



These guards can be freely mixed. **Note** that there is no *syntactic* distinction between these guards - they are all expressions. However they may be distinguished at the semantic level.

A mutex predicate allows the user to specify either that all operations of the class are to be executed mutually exclusive, or that a list of operations are to be executed mutually exclusive to each other. Operations that appear in one mutex predicate are allowed to appear in other mutex predicates as well, and may also be used in the usual permission predicates. Each mutex predicate will implicitly be translated to permission predicates using history guards for each operation mentioned in the name list. For instance,

```
sync
  mutex (opA, opB);
  mutex (opB, opC, opD);
  per opD => someVariable > 42;
```

would be translated to the following permission predicates:

```
sync
  per opA => #active (opB) = 0;
  per opB => #active (opA) = 0 and
             #active (opC) + #active (opD) = 0;
  per opC => #active (opB) + #active (opD) = 0;
  per opD => #active (opB) + #active (opC) = 0 and
             someVariable > 42;
```

Note that it is only permitted to have one “stand-alone” permission predicate for each operation. It is also important to note that if permission predicates are made over operations that are overloaded (see Section 13.3.1) then it will incorporate all of their history counters as the same operation. The **#active** operator is explained below.

A **mutex(all)** constraint specifies that all of the operations specified in that class *and any superclasses* are to be executed mutually exclusively.

### 14.1.1 History guards

**Semantics:** A history guard is a guard which depends on the sequence of earlier invocations of the operations of the object expressed in terms of history expressions (see section 6.23). History expressions denotes the number of activations and completions of the operations, given as functions

**#act** and **#fin**, respectively.

**#act**: operation name  $\rightarrow \mathbb{N}$

**#fin**: operation name  $\rightarrow \mathbb{N}$



Furthermore, a derived function **#active** is available such that **#active** (A) = **#act** (A) - **#fin** (A), giving the number of currently active instances of A. Another history function - **#req** - is defined in section 14.1.3.

**Examples:** Consider a Web server that is capable of supporting 10 simultaneous connections and can buffer a further 100 requests. In this case we have one instance variable, representing the mapping from URLs to local filenames:

**instance variables**

```
site_map : map URL to Filename := {|->}
```

The following operations are defined in this class (definitions omitted for brevity):

ExecuteCGI:	URL ==> File	Execute a CGI script on the server
RetrieveURL:	URL ==> File	Transmit a page of html
UploadFile:	File * URL ==> ()	Upload a file onto the server
ServerBusy:	() ==> File	Transmit a “server busy” page
DeleteURL:	URL ==> ()	Remove an obsolete file

Since the server can support only 10 simultaneous connects, we can only permit an execute or retrieve operation to be activated if the number already active is less than 10:

```
per RetrieveURL => #active(RetrieveURL) +
                  #active(ExecuteCGI) < 10;
per ExecuteCGI  => #active(RetrieveURL) +
                  #active(ExecuteCGI) < 10;
```

## 14.1.2 The object state guard

**Semantics:** The object state guard is a boolean expression which depends on the values of one (or more) instance variable(s) of the object itself. Object state guards differ from operation pre-conditions in that a call to an operation whose permission predicate is false results in the caller blocking until the predicate is satisfied, whereas a call to an operation whose pre-condition is false means the operation’s behaviour is unspecified.

**Examples:** Using the web server example again, we can only allow file removal if some files already exist:

```
per DeleteURL    => dom site_map <> {}
```

Constraints for safe execution of the operations Push and Pop in a stack object can be expressed using an object state guard as:



```
per Push => length < maxsize;
per Pop => length > 0
```

where `maxsize` and `length` are instance variables of the stack object.

It is often possible to express such constraints as a consequence of the history, for example the empty state of the stack:

```
length = 0 <=> #fin(Push) = #fin(Pop)
```

However, the size is a property which is better regarded as a property of the particular stack instance, and in such cases it is more elegant to use available instance variables which store the effects of history.

### 14.1.3 Queue condition guards

**Semantics:** A queue condition guard acts on requests waiting in the queues for the execution of the operations. This requires use of a third history function `#req` such that `#req(A)` counts the number of messages which have been received by the object requesting execution of operation `A`. Again it is useful to introduce the function `#waiting` such that: `#waiting(A) = #req(A) - #act(A)`, which counts the number of items in the queue.

**Examples:** Once again, with the web server we can only activate the `ServerBusy` operation if 100 or more connections are waiting:

```
per ServerBusy => #waiting(RetrieveURL) +
                  #waiting(ExecuteCGI) >= 100;
```

The most important use of such expressions containing queue state functions is for expressing priority between operations. The protocol specified by:

```
per B => #waiting(A) = 0
```

gives priority to waiting requests for activation of `A`. There are, however, many other situations when operation dispatch depends on the state of waiting requests. Full description of the queuing requirements to allow specification of operation selection based on request arrival times or to describe ‘shortest job next’ behaviour will be a future development.

Note that `#req(A)` have value 1 at the time of evaluation of the permission predicate for the first invocation of operation `A`. That is,





```
per A => #req(A) = 0
```

would always block.

### 14.1.4 Evaluation of Guards

Using the previous example, consider the following situation: the web server is handling 10 `RetrieveURL` requests already. While it is dealing with these requests, two further `RetrieveURL` requests (from objects  $O_1$  and  $O_2$ ) and one `ExecuteCGI` request (from object  $O_3$ ) are received. The permission predicates for these two operations are false since the number of active `RetrieveURL` operations is already 10. Thus these objects block.

Then, one of the active `RetrieveURL` operations reaches completion. The permission predicate so far blocking  $O_1$ ,  $O_2$  and  $O_3$  will become “true” simultaneously. This raises the question: which object is allowed to proceed? Or even all of them?

Guard expressions are only reevaluated when an event occurs (in this case the completion of a `RetrieveURL` operation). In addition to that the test of a permission predicate by an object and its (potential) activation is an atomic operation. This means, that when the first object evaluates its guard expression, it will find it to be true and activate the corresponding operation (`RetrieveURL` or `ExecuteCGI` in this case). The other objects evaluating their guard expressions afterwards will find that `#active(RetrieveURL) + #active(ExecuteCGI) = 10` and thus remain blocked. *Which object is allowed to evaluate the guard expression first is undefined.*

It is important to understand that the guard expression need only evaluate to **true** at the time of the activation. In the example as soon as  $O_1$ ,  $O_2$  or  $O_3$ ’s request is activated its guard expression becomes false again.

## 14.2 Inheritance of Synchronization Constraints

Synchronization constraints specified in a superclass are inherited by its subclass(es). The manner in which this occurs depends on the kind of synchronization.

### 14.2.1 Mutex constraints

Mutex constraints from base classes and derived classes are simply added. If the base class and derived class have the mutex definitions  $M_A$  and  $M_B$ , respectively, then the derived class simply has both mutex constraints  $M_A$ , and  $M_B$ . The binding of operation names to actual operations is always performed in the class where the constraint is defined. Therefore a **mutex(all)** constraint defined in a superclass and inherited by a subclass only makes the operations from the base class mutually exclusive and does not affect operations of the derived class.



Inheritance of mutex constraints is completely analogous to the inheritance scheme for permission predicates. Internally mutex constraints are always expanded into appropriate permission predicates which are added to the existing permission predicates as a conjunction. This inheritance scheme ensures that the result (the final permission predicate) is the same, regardless of whether the mutex definitions are expanded in the base class and inherited as permission predicates or are inherited as mutex definitions and only expanded in the derived class.

The intention for inheriting synchronization constraints in the way presented is to ensure, that any derived class at least satisfies the constraints of the base class. In addition to that it must be possible to strengthen the synchronization constraints. This can be necessary if the derived class adds new operations as in the following example:

```
class A
operations

  writer: () ==> ()
  writer() == is not yet specified

  reader: () ==> ()
  reader() == is not yet specified

  sync
  per reader => #active(writer) = 0;
  per writer => #active(reader, writer) = 0;
end A

class B is subclass of A
operations

  newWriter: () ==> ()
  newWriter() == is not yet specified

  sync
  per reader => #active(newWriter) = 0;
  per writer => #active(newWriter) = 0;
  per newWriter => #active(reader, writer, newWriter) = 0;

end B
```

Class A implements reader and writer operations with the permission predicates specifying the multiple readers-single writer protocol. The derived class B adds `newWriter`. In order to ensure deterministic behaviour B also has to add permission predicates for the inherited operations.

The actual permission predicates in the derived class are therefore:



```
per reader => #active(writer)=0 and #active(newWriter)=0;  
per writer => #active(reader, writer)=0 and #active(newWriter)=0;  
per newWriter => #active(reader, writer, newWriter)=0;
```

A special situation arises when a subclass overrides an operation from the base class. The overriding operation is treated as a new operation. It has no permission predicate (and in particular inherits none) unless one is defined in the subclass.

The semantics of inheriting mutex constraints for overridden operations is completely analogous: newly defined overriding operations are not restricted by mutex definitions for equally named operations in the base class. The **mutex(all)** shorthand makes all inherited and locally defined operations mutually exclusive. Overridden operations (defined in a base class) are not affected. In other words, all operations, that can be called with an unqualified name (“locally visible operations”) will be mutex to each other.

# Chapter 15

## Threads (VDM++ and VDM-RT)

Objects instantiated from a class with a *thread* part are called *active* objects. The scope of the instance variables and operations of the current class is considered to extend to the thread specification. Note that from a tool perspective the thread for the expression a user would like to evaluate in relation to a VDM model is called a debug thread. This thread has a special role in the sense that when it is finished the entire execution is completed (and thus all other threads ready to be scheduled in or running will be thrown away and aborted). If a session where a series of expressions are being evaluated this is not true (in this case the other threads will be continued when the next expression is executed, see [?] for more details about “sessions”). Thus if one would like to ensure a specific number of such other threads to be completed before stopping the execution one needs to block the debug thread using a synchronisation as explained in Chapter 14.

**Syntax:** thread definitions = ‘**thread**’, [ thread definition ] ;

thread definition = periodic thread definition  
                          | procedural thread definition ;

periodic thread definition = periodic obligation  
                                  | sporadic obligation ;

Subclasses inherit threads from superclasses. If a class inherits from several classes only one of these may declare its own thread (possibly through inheritance). Furthermore, explicitly declaring a thread in a subclass will override any inherited thread.

### 15.1 Periodic Thread Definitions (VDM-RT)

The periodic obligation can be regarded as the way of describing repetitive activities in a class. As the ‘period’ implies a explicit notion of time, this construct is only available in VDM-RT.

**Syntax(VDM-RT):** Time is explicit in VDM-RT, using a discrete clock with a 1 nsec resolution, where the periodic obligation looks like:

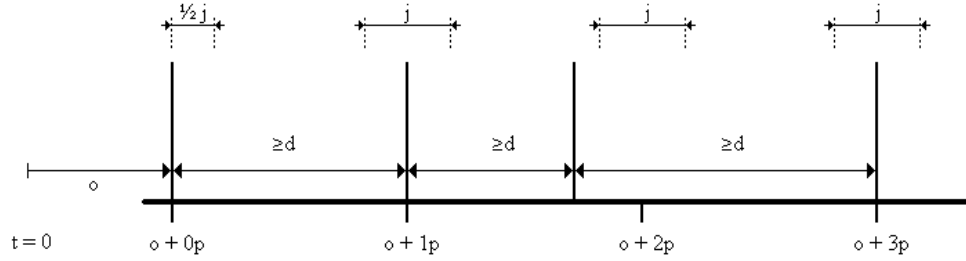


Figure 15.1: Period (p), jitter (j), delay (d) and offset (o)

periodic obligation = **'periodic'**, '(', 4 \* expression, ')', '(', name, ')';

**Semantics (VDM-RT):** The type of the expressions should all yield a natural number (as we use a natural number valued wall-clock in the VDM-RT interpreter), otherwise a run-time error will occur. Note that the evaluation of the expressions also causes time to elapse whenever the **'start'** or **'start\_list'** statement is executed. The expressions all denote a time value with a resolution of 1 nsec. For each periodic obligation, four different numbers are used. They are, in order of appearance (also illustrated in Figure 15.1):

1. **period:** This is a non-negative, non-zero value that describes the length of the time interval between two adjacent events in a strictly periodic event stream (where jitter = 0). Hence, a value of 1E9 denotes a period of 1 second.
2. **jitter:** This is a non-negative value that describes the amount of time variance that is allowed around a single event. We assume that the interval is balanced  $[-j, j]$ . Note that jitter is allowed to be bigger than the period to characterize so-called event bursts.
3. **delay:** This is a non-negative value smaller than the period which is used to denote the minimum inter arrival distance between two adjacent events.
4. **offset:** This is a non-negative value which is used to denote the absolute time value at which the first period of the event stream starts. Note that the first event occurs in the interval  $[\text{offset}, \text{offset} + \text{jitter}]$ .

Given a defined time resolution  $\Delta T$ , a thread with a periodic obligation invokes the mentioned operation at the beginning of each time interval with length *period*. This creates the periodic execution of the operation simulating the discrete equivalent of continuous relations which have to be maintained between instance variables, parameter values and possibly other external values obtained through operation invocations. It is not possible to dynamically change the length of the interval.

Periodic obligations are intended to describe e.g. analogue physical relations between values in formulas (e.g. transfer functions) and their discrete event simulation. It is a requirement on the implementation to guarantee that the execution time of the operation is at least smaller



than the used periodic time length. If other operations are present the user has to guarantee that the fairness criteria for the invocation of these other operations are maintained by reasoning about the time slices used internally and available for external invocations.

Note that a periodic thread is *neither* created *nor* started when an instance of the corresponding class is created. Instead, as with procedural threads, ‘**start**’ (or ‘**start\_list**’) statements should be used with periodic threads.

**Examples:** Consider a timer class which periodically increments its clock in its own thread. It provides operations for starting, and stopping timing, and reading the current time.

```
class Timer
values
  PERIOD : nat = 1000
```

The Timer has two instance variables the current time and a flag indicating whether the Timer is active or not (the current time is only incremented if the Timer is active).

```
instance variables
  curTime : nat := 0;
  acti    : bool := false;
```

The Timer provides straightforward operations which need no further explanation.

```
operations
public Start : () ==> ()
Start() ==
  (acti := true;
   curTime := 0);

public Stop : () ==> ()
Stop() ==
  acti := false;

public GetTime : () ==> nat
GetTime() ==
  return curTime;

IncTime: () ==> ()
IncTime() ==
  if acti
  then curTime := curTime + 100;
```



The Timer's thread ensures that the current time is incremented. The period with which this is done is 1000 time units (nanoseconds). The allowed jitter is 10 time units and the minimal distance between two instances is 200 time units and finally no offset has been used.

```
thread
  periodic (PERIOD, PERIOD/100, PERIOD/5, 0) (IncTime)
end Timer
```

## 15.2 Sporadic Thread Definitions (VDM-RT)

The sporadic obligation can be regarded as the way of describing stochastic activities in a class.

**Syntax: (VDM-RT):**

sporadic obligation = '**sporadic**', '(', 3 \* expression, ')', '(', name, ')';

**Semantics: (VDM-RT)** The type of the expressions should all yield a natural number (as we use a natural number valued wall-clock in the VDM-RT interpreter), otherwise a run-time error will occur. Note that the evaluation of the expressions also causes time to elapse whenever the '**start**' or '**start\_list**' statement is executed. The expressions all denote a time value with a resolution of 1 nsec. For each sporadic obligation, three different numbers are used. They are, in order of appearance:

1. **delay:** This value is used to denote the minimum inter arrival distance between two adjacent thread invocations.
2. **bound:** This value, greater than **delay**, is the maximum inter arrival distance between two adjacent thread invocations.
3. **offset:** This is a non-negative value which is used to denote the absolute time value at which the first period starts, randomly in the interval  $[offset, offset + bound]$ .

Given these definitions, and assuming the last thread was invoked at  $t_0$  then the next invocation is randomly scheduled in the interval  $[t_0 + delay, t_0 + bound]$ .

**Examples:** Analogous to the example in '**periodic**' for the `Timer` class, the sporadic definition could be as follows:

```
class Timer
  ...
```



```

thread
  sporadic (100, 1000, 0) (IncTime)

end Timer

```

### 15.3 Procedural Thread Definitions (VDM++ and VDM-RT)

A procedural thread provides a mechanism to explicitly define the external behaviour of an active object through the use of *statements*, which are executed when the object is started (see section 12.14).

**Syntax:** procedural thread definition = statement ;

**Semantics:** A procedural thread is scheduled for execution following the application of a start statement to the object owning the thread. The statements in the thread are then executed sequentially, and when execution of the statements is complete, the thread dies. Synchronization between multiple threads is achieved using permission predicates on shared objects.

**Examples:** The example below demonstrates procedural threads by using them to compute the factorial of a given integer concurrently.

```

class Factorial

  instance variables
    result : nat := 5;
  operations

  public factorial : nat ==> nat
  factorial(n) ==
    if n = 0
    then return 1
    else (dcl m : Multiplier;
          m := new Multiplier();
          m.calculate(1,n);
          start (m);
          result:= m.giveResult();
          return result
        )

end Factorial

```





```
class Multiplier

instance variables
  i : nat1;
  j : nat1;
  k : nat1;
  result : nat1

operations

public calculate : nat1 * nat1 ==> ()
calculate (first, last) ==
  (i := first; j := last);

doit : () ==> ()
doit() ==
  ( if i = j
    then result := i
    else (dcl p : Multiplier;
         dcl q : Multiplier;
         p := new Multiplier();
         q := new Multiplier();
         start(p);
         start(q);
         k := (i + j) div 2;
         -- division with rounding down
         p.calculate(i,k);
         q.calculate(k+1,j);
         result := p.giveResult() * q.giveResult ()
        )
  );

public giveResult : () ==> nat1
giveResult() ==
  return result;

sync
-- cyclic constraints allowing only the
-- sequence calculate; doit; giveResult

per doit => #fin (calculate) > #act(doit);
per giveResult => #fin (doit) > #act (giveResult);
```



```
per calculate => #fin (giveResult) = #act (calculate)

thread
  doit();

end Multiplier
```



# Chapter 16

## Trace Definitions

In order to automate the testing process VDM-10 contains a notation enabling the expression of the traces that one would like to have tested exhaustively. Such traces are used to express combinations of sequences of operations that wish to be tested in all possible combinations. In a sense this is similar to model checking limitations except that this is done with real and not symbolic values. However, errors in test cases are filtered away so other test cases with the same prefix will be skipped automatically.

**Syntax:** traces definitions = **'traces'**, [ named trace, { **';**', named trace } ] ;

named trace = identifier, { **'/'**, identifier }, **':'**, trace definition list ;

trace definition list = trace definition term, { **';**', trace definition term } ;

trace definition term = trace definition, { **'|'**, trace definition } ;

trace definition = trace binding definition  
| trace repeat definition ;

trace binding definition = trace let def binding  
| trace let best binding ;

trace let def binding = **'let'**, local definition, { **'/'**, local definition },  
**'in'**, trace definition ;

trace let best binding = **'let'**, multiple bind, [ **'be'**, **'st'**, expression ],  
**'in'**, trace definition ;

trace repeat definition = trace core definition, [ trace repeat pattern ] ;



```

trace repeat pattern = '*'
                    | '+'
                    | '?'
                    | '{', numeric literal, [ ',', numeric literal, '}' ] ;

trace core definition = trace apply expression
                    | trace concurrent expression
                    | trace bracketed expression ;

trace apply expression = call statement ;

trace concurrent expression = '| |', '(', trace definition,
                             ',', trace definition,
                             { ',', trace definition }, ')' ;

trace bracketed expression = '(', trace definition list, ')' ;

```

**Semantics:** Semantically the trace definitions provided in a class have no effect. These definitions are simply used to enhance testing of a VDM specification using principles from combinatorial testing (also called all-pairs testing). So each trace definition can be considered as a regular expression describing the test sequences in which different operations should be executed to test the VDM specification. Inside the trace definitions, bindings may appear and for each possible binding a particular test case can be automatically derived. So one trace definition expands into a set of test cases. In this sense a test case is a sequence of operation calls executed after each other. Between each test case the VDM specification is initialised so they become entirely independent. From a static semantics perspective it is important to note that the expressions used inside trace definitions must be executed in the expansion process. This means that it cannot directly refer to instance variables, because these could be changed during the execution.

So here it makes sense to explain what kind of expansion the different kinds of trace definitions gives rise to.

The *trace definition lists* simply use a semicolon (“;”) and this results in sequencing between the *trace definition terms* used inside it.

In the *trace definition term* it is possible to introduce alternatives using the bar (“|”) operator. This results in test cases for all alternatives.

The *trace binding definition* exists in two forms where the *trace let def binding* simply enables the binding introduced to be used after the ‘**in**’ in the same way as in let-expressions. Alternatively the *trace let best binding* can be used and this will expand to test cases with all the different possible bindings.

The *trace repeat definition* is used to introduce the possibility of having repetitions of the operation calls used in the trace. The different kinds of repeat patterns have the following meanings:



- ‘\*’ means 0 to n occurrences (n is tool specific).
- ‘+’ means 1 to n occurrences (n is tool specific).
- ‘?’ means 0 or 1 occurrences.
- ‘{’, n, ‘}’ means n occurrences.
- ‘{’, n, ‘,’ m ‘}’ means between n and m occurrences.

The *trace core definitions* have three possibilities. These are ordinary operation calls, trace concurrency expressions and bracketed trace definitions respectively. The *trace concurrency expressions* are similar to the *nondeterministic statements* in the sense that the trace definition lists inside it will be executed in all possible permutations of the elements. This is particular useful for concurrent VDM++ models where potential deadlocks can occur under some circumstances.

**Examples:** In an example like the one below test cases will be generated in all possible combination starting with a call of `Reset` followed by one to four `Pushes` of values onto the stack followed again by one to three `Pops` from the stack.

```
class Stack

instance variables
    stack : seq of int := [];

operations

    public Reset : () ==> ()
    Reset () ==
        stack := [];

    public Pop : () ==> int
    Pop() ==
        def res = hd stack in
            (stack := tl stack;
             return res)
    pre stack <> []
    post stack~ = [RESULT] ^ stack;

    public Push: int ==> ()
    Push(elem) ==
        stack := stack ^ [elem];

    public Top : () ==> int
```



```
Top() ==  
    return (hd stack);  
  
end Stack  
class UseStack  
  
instance variables  
  
    s : Stack := new Stack();  
  
traces  
  
    PushBeforePop : s.Reset();  
                    (let x in set {1,2} in s.Push(x)) {1,4};  
                    s.Pop() {1,3}  
  
end UseStack
```

# Appendix A

## The Syntax of the VDM Languages

This appendix specifies the complete syntax for the VDM languages.

### A.1 VDM-SL Document

```
document = module, { module }  
          | definition block, { definition block } ;
```

#### A.1.1 Modules

This entire subsection is not present in the current version of the VDM-SL 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  
                | export values signature  
                | export functions signature  
                | export operations signature ;  
  
export types signature = 'types', type export,  
                        { ';', type export }, [ ';' ] ;  
  
type export = [ 'struct' ], name ;  
  
export values signature = 'values', value signature,  
                        { ';', value signature }, [ ';' ] ;  
  
value signature = name list, ':', type ;  
  
export functions signature = 'functions', function signature,  
                        { ';', function signature }, [ ';' ] ;  
  
function signature = name list, [ type variable list ], ':',  
                    function type ;  
  
export operations signature = 'operations', operation signature,  
                        { ';', operation signature }, [ ';' ] ;  
  
operation signature = name list, ':', operation type ;
```



## A.2 VDM++ and VDM-RT Document

document = ( class | system ), { class | system } ;

## A.3 System (VDM-RT)

system = **'system'**, identifier,  
[ class body ],  
**'end'**, identifier ;

### A.3.1 Classes

class = **'class'**, identifier, [ inheritance clause ],  
[ class body ],  
**'end'**, identifier ;

inheritance clause = **'is subclass of'**, identifier, { ',', identifier } ;

## A.4 Definitions

class body = definition block, { definition block } ;

module body = **'definitions'**, definition block, { definition block } ;

definition block = type definitions  
| state definition  
| value definitions  
| function definitions  
| operation definitions  
| instance variable definitions  
| synchronization definitions  
| thread definitions  
| traces definitions ;

### A.4.1 Type Definitions

type definitions = **'types'**, [ access type definition ,  
{ **';**', access type definition }, [ **';**' ] ] ;

access type definition = ([ access ], [ **'static'** ] | [ **'static'** ], [ access ]),  
type definition ;



The access part is only possible in VDM++ and VDM-RT.

```
access = 'public'
        | 'private'
        | 'protected' ;
```

```
type definition = identifier, '=', type, [ invariant ], [ eq clause ], [ ord clause ]
                | identifier, ': : ', field list, [ invariant ], [ eq clause ], [ ord clause ] ;
```

```
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 = set0 type
          | set1 type ;
```



```
set0 type = 'set of', type ;
set1 type = 'set1 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 ;
invariant = 'inv', invariant initial function ;
invariant initial function = pattern, '==', expression ;
eq clause = 'eq', pattern, '=', pattern, '==', expression ;
ord clause = 'ord', pattern, '<', pattern, '==', expression ;
```

#### A.4.2 The VDM-SL State Definition

```
state definition = 'state', identifier, 'of', field list,
                  [ invariant ], [ initialisation ], 'end', [ ';' ] ;
initialisation = 'init', invariant initial function ;
```



### A.4.3 Value Definitions

value definitions = **'values'**, [ access value definition,  
                  { **';**', access value definition }, [ **';**' ] ] ;

access value definition = [ access ], value definition ;

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

### A.4.4 Function Definitions

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

access function definition = [ access ], 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'**, measure body ] ;

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

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 pair, { ',', identifier type pair } ;

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

parameters list = parameters, { parameters } ;

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

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

measure body = expression
              | 'is not yet specified' ;
```

#### A.4.5 Operation Definitions

```
operation definitions = 'operations', [ access operation definition,
                                     { ';', access operation definition }, [ ';' ] ] ;

access operation definition = { 'pure'
                              | 'async'
                              | access
                              | 'static' },
                             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 ;

operation body = statement
                | 'is subclass responsibility'
                | 'is not yet specified' ;

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.4.6 Instance Variable Definitions (VDM++ and VDM-RT)

```
instance variable definitions = 'instance', 'variables',
                               [ instance variable definition,
                               { ';', instance variable definition } ] ;

instance variable definition = access assignment definition
                             | invariant definition ;

access assignment definition = ([ access ], [ 'static' ] | [ 'static' ], [ access ]),
                              assignment definition ;

invariant definition = 'inv', expression ;
```



#### A.4.7 Synchronization Definitions (VDM++ and VDM-RT)

```

synchronization definitions = 'sync', [ synchronization ] ;

synchronization = permission predicates ;

permission predicates = permission predicate,
                      { ';', permission predicate } ;

permission predicate = 'per', name, '=>', expression
                    | mutex predicate ;

mutex predicate = 'mutex', '(', ('all' | name list), '(' ;

```

#### A.4.8 Thread Definitions (VDM++ and VDM-RT)

```

thread definitions = 'thread', [ thread definition ] ;

thread definition = periodic thread definition
                  | procedural thread definition ;

periodic thread definition = periodic obligation
                           | sporadic obligation ;

```

For VDM-RT where time is explicit, it looks like:

```

periodic obligation = 'periodic', '(', 4 * expression, '(', name, '(' ;

sporadic obligation = 'sporadic', '(', 3 * expression, '(', name, '(' ;

```

For both VDM++ and VDM-RT, we can define:

```

procedural thread definition = statement ;

```

#### A.4.9 Trace Definitions

```

traces definitions = 'traces', [ named trace, { ';', named trace } ] ;

named trace = identifier, { '/', identifier }, '::', trace definition list ;

trace definition list = trace definition term, { ';', trace definition term } ;

trace definition term = trace definition, { '|', trace definition } ;

trace definition = trace binding definition
                  | trace repeat definition ;

```





```
trace binding definition = trace let def binding
                        | trace let best binding ;

trace let def binding = 'let', local definition, { ',', local definition },
                        'in', trace definition ;

trace let best binding = 'let', multiple bind, [ 'be', 'st', expression ],
                        'in', trace definition ;

trace repeat definition = trace core definition, [ trace repeat pattern ] ;

trace repeat pattern = '*'
                    | '+'
                    | '?'
                    | '{', numeric literal, [ ',', numeric literal ], '}' ;

trace core definition = trace apply expression
                    | trace concurrent expression
                    | trace bracketed expression ;

trace apply expression = call statement ;

trace concurrent expression = '| |', '(', trace definition,
                             ',', trace definition,
                             { ',', trace definition }, ')' ;

trace bracketed expression = '(', trace definition list, ')' ;
```

## A.5 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
- | narrow expression
- | new expression
- | self expression
- | threadid expression
- | general is expression
- | undefined expression
- | precondition expression
- | isofbaseclass expression
- | isofclass expression
- | samebaseclass expression
- | sameclass expression
- | act expression
- | fin expression
- | active expression
- | req expression
- | waiting expression
- | time expression
- | name
- | old name
- | symbolic literal ;

### A.5.1 Bracketed Expressions

bracketed expression = ‘(’, expression, ‘)’ ;



### A.5.2 Local Binding Expressions

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

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

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

### A.5.3 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.5.4 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
	sequence reverse
	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**' ;  
 sequence reverse = '**reverse**' ;  
 distributed sequence concatenation = '**conc**' ;  
 map domain = '**dom**' ;  
 map range = '**rng**' ;  
 distributed map merge = '**merge**' ;  
 map inverse = '**inverse**', expression ;



### A.5.5 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 = '\*' ;



## APPENDIX A. THE SYNTAX OF THE VDM LANGUAGES

---

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.5.6 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.5.7 The Iota Expression

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

### A.5.8 Set Expressions

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

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

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

### A.5.9 Sequence Expressions

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

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

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

### A.5.10 Map Expressions

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

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

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



### A.5.11 The Tuple Constructor Expression

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

### A.5.12 Record Expressions

record constructor = `'mk_'`<sup>1</sup> name, `'('`, [ expression list ], `)'` ;

record modifier = `'mu'`, `'('`, expression, `' '`,  
record modification,  
{ `' '`, record modification }, `)'` ;

record modification = identifier, `'|->'`, expression ;

### A.5.13 Apply Expressions

apply = expression, `'('`, [ expression list ], `)'` ;

field select = expression, `'.'`, identifier ;

tuple select = expression, `'.#'`, numeral ;

function type instantiation = name, `'['`, type, { `' '`, type }, `']'` ;

### A.5.14 The Lambda Expression

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

### A.5.15 The narrow Expression

narrow expression = `'narrow_'`, `'('`, expression, `' '`, type, `)'` ;

### A.5.16 The New Expression (VDM++ and VDM-RT)

new expression = `'new'`, name, `'('`, [ expression list ], `)'` ;

### A.5.17 The Self Expression (VDM++ and VDM-RT)

self expression = `'self'` ;

---

<sup>1</sup>**Note:** no delimiter is allowed





### A.5.18 The Threadid Expression (VDM++ and VDM-RT)

threadid expression = **'threadid'** ;

### A.5.19 The Is Expression

general is expression = is expression  
                                  | type judgement ;

is expression = **'is\_'**,<sup>2</sup>( name | basic type), **'(, expression, ')**' ;

type judgement = **'is\_'**, **'(, expression, ', type, ')**' ;

### A.5.20 The Undefined Expression

undefined expression = **'undefined'** ;

### A.5.21 The Precondition Expression

precondition expression = **'pre\_'**, **'(, expression list, ')**' ;

### A.5.22 Base Class Membership (VDM++ and VDM-RT)

isofbaseclass expression = **'isofbaseclass'**, **'(, name, ', expression, ')**' ;

### A.5.23 Class Membership (VDM++ and VDM-RT)

isofclass expression = **'isofclass'**, **'(, name, ', expression, ')**' ;

### A.5.24 Same Base Class Membership (VDM++ and VDM-RT)

samebaseclass expression = **'samebaseclass'**, **'(, expression, ', expression, ')**' ;

### A.5.25 Same Class Membership (VDM++ and VDM-RT)

sameclass expression = **'sameclass'**, **'(, expression, ', expression, ')**' ;

---

<sup>2</sup>**Note:** no delimiter is allowed



### A.5.26 History Expressions (VDM++ and VDM-RT)

act expression = `#act`, `(`, name list, `)` ;  
 fin expression = `#fin`, `(`, name list, `)` ;  
 active expression = `#active`, `(`, name list, `)` ;  
 req expression = `#req`, `(`, name list, `)` ;  
 waiting expression = `#waiting`, `(`, name list, `)` ;

### A.5.27 Time Expressions (VDM-RT)

time expression = `time` ;

### A.5.28 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, `)` ;

## A.7 Statements

statement = let statement  
           | let be statement  
           | def statement  
           | block statement  
           | general assign statement  
           | if statement  
           | cases statement



- | sequence for loop
- | set for loop
- | index for loop
- | while loop
- | nondeterministic statement
- | call statement
- | specification statement
- | start statement
- | start list statement
- | stop statement
- | stop list statement
- | duration statement
- | cycles statement
- | return statement
- | always statement
- | trap statement
- | recursive trap statement
- | exit statement
- | error statement
- | 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'**, multiple bind, [ **'be'**, **'st'**, expression ], **'in'**,  
                  statement ;

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

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

### A.7.2 Block and Assignment Statements

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



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

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

general assign statement = assign statement  
                              | multiple assign statement ;

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'**,  
                      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

In VDM-SL a call statement looks like:

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

In VDM++ and VDM-RT a call statement looks like:

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

object designator = name  
| self expression  
| new expression  
| object field reference  
| object apply ;

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

object apply = object designator, '(', [ expression list ], ') ' ;

return statement = **return**, [ expression ] ;

### A.7.7 The Specification Statement

specification statement = '[', implicit operation body, ']' ;

### A.7.8 Start and Start List Statements (VDM++ and VDM-RT)

start statement = **start**, '(', expression, ') ' ;

start list statement = **startlist**, '(', expression, ') ' ;

### A.7.9 Stop and Stop List Statements (VDM++ and VDM-RT)

stop statement = **stop**, '(', expression, ') ' ;

stop list statement = **stoplist**, '(', expression, ') ' ;



### A.7.10 The Duration and Cycles Statements (VDM-RT)

```
duration statement = 'duration', '(', expression, ')',
                    statement ;

cycles statement = 'cycles', '(', expression, ')',
                  statement ;
```

### A.7.11 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.7.12 The Error Statement

```
error statement = 'error' ;
```

### A.7.13 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 munion pattern
         | tuple pattern
         | object pattern
         | record pattern ;
```



pattern identifier = identifier | ‘-’ ;

match value = ‘(’, expression, ‘)’  
              | symbolic literal ;

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 munion pattern = pattern, ‘**munion**’, pattern ;

tuple pattern = ‘mk\_’, ‘(’, pattern, ‘,’, pattern list, ‘)’ ;

record pattern = ‘mk\_’,<sup>3</sup> name, ‘(’, [ pattern list ], ‘)’ ;

object pattern = ‘**obj\_**’, identifier, ‘(’, [ field pattern list ], ‘)’<sup>4</sup>;

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

field pattern = identifier, ‘|->’, pattern ;

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

<sup>3</sup>**Note:** no delimiter is allowed

<sup>4</sup>**Note:** object pattern is only be used in VDM++ and VDM-RT



### A.8.2 Bindings

```
pattern bind = pattern | bind ;

bind = set bind | seq bind | type bind ;

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

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

type bind = pattern, ':', type ;

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

multiple bind = multiple set bind
               | multiple seq bind
               | multiple type bind ;

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

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

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

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





# Appendix B

## Lexical Specification

### B.1 Characters

The characters that comprise a valid VDM specification are defined in terms of Unicode codepoints. The actual character encoding of a VDM source file (for example UTF-8, ISO-Latin-1 or Shift-JIS) is not defined, and the tool support is responsible for converting whatever encoding is used into Unicode during the parse of the file.

All VDM keywords and delimiter tokens are composed of characters from the Basic Latin block (“ASCII” codepoints less than U+0080). On the other hand, user identifiers (variable names, function names and so on) can be composed of a rich variety of Unicode codepoints, reflecting the need for fully internationalized specifications.

All Unicode codepoints have a “category”. Certain categories are entirely excluded from the set of codepoints that are permitted in identifiers. This prevents, say, punctuation characters from being used. On the other hand, to provide a degree of compatibility with the original VDM ISO standard, and for backward compatibility, there are different rules for the formation of user identifiers that only use ASCII characters. For example, the underscore is permitted in identifiers (U+005F), even though this is in the connecting punctuation category, which would not normally be allowed.

See <http://www.fileformat.info/info/unicode/category/index.htm> for more information about categories.



---

initial letter:

if       codepoint < U+0100  
then     Any character in categories Ll, Lm, Lo, Lt, Lu or U+0024 (a dollar sign)  
else     Any character except categories Cc, Zl, Zp, Zs, Cs, Cn, Nd, Pc

following letter:

if       codepoint < U+0100  
then     Any character in categories Ll, Lm, Lo, Lt, Lu, Nd or U+0024 (a dollar sign)  
          or U+005F (underscore) or U+0027 (apostrophe)  
else     Any character except categories Cc, Zl, Zp, Zs, Cs, Cn

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
---	---	---	---	---	---	---	---

---

Table B.1: 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 = '#act' | '#active' | '#fin' | '#req' | '#waiting' | 'abs'
         | 'all' | 'always' | 'and' | 'as' | 'async' | 'atomic' | 'be'
         | 'bool' | 'by' | 'card' | 'cases' | 'char' | 'class'
         | 'comp' | 'compose' | 'conc' | 'cycles' | 'dcl' | 'def'
         | 'definitions' | 'dinter' | 'div' | 'dlmodule' | 'do'
         | 'dom' | 'dunion' | 'duration' | 'elems' | 'else' | 'elseif'
         | 'end' | 'eq' | 'error' | 'errs' | 'exists' | 'exists1' | 'exit'
         | 'exports' | 'ext' | 'false' | 'floor'
         | 'for' | 'forall' | 'from' | 'functions' | 'hd' | 'if' | 'in'
         | 'inds' | 'inmap' | 'instance' | 'int' | 'inter'
         | 'imports' | 'init' | 'inv' | 'inverse' | 'iota' | 'is'
         | 'isofbaseclass' | 'isofclass' | 'lambda' | 'len' | 'let'
         | 'map' | 'measure' | 'merge' | 'mod' | 'module' | 'mu'
         | 'munion' | 'mutex' | 'nat' | 'nat1' | 'new' | 'nil' | 'not' | 'of'
         | 'operations' | 'or' | 'ord' | 'others' | 'per' | 'periodic' | 'post'
         | 'power' | 'pre' | 'private' | 'protected' | 'psubset'
         | 'public' | 'pure' | 'rat' | 'rd' | 'real' | 'rem' | 'renamed'
         | 'responsibility' | 'return' | 'reverse' | 'rng'
         | 'samebaseclass' | 'sameclass' | 'self' | 'seq' | 'seq1'
         | 'set' | 'set1' | 'skip' | 'specified' | 'sporadic' | 'st' | 'start'
         | 'startlist' | 'state' | 'stop' | 'stoplist'
         | 'struct' | 'subclass' | 'subset' | 'sync'
         | 'system' | 'then' | 'thread' | 'threadid' | 'time' | 'tixe'
         | 'tl' | 'to' | 'token' | 'traces' | 'trap' | 'true' | 'types'
         | 'undefined' | 'union' | 'uselib' | 'values'
         | 'variables' | 'while' | 'with' | 'wr' | 'yet' | 'RESULT' ;
```

identifier = initial letter, { following letter } ;

Note that in VDM-RT the CPU and BUS classes are reserved and cannot be redefined by the user. These two predefined classes contain the functionality described in Section 13.3 above.

All identifiers beginning with one of the reserved prefixes are reserved: **init\_**, **inv\_**, **is\_**, **mk\_**, **post\_** and **pre\_**.

type variable identifier = '@', identifier ;



```
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, [ '.', numeral ], [ exponent ] ;

hexadecimal literal = ( '0x' | '0X' ), hexadecimal digit, { hexadecimal digit } ;

boolean literal = 'true' | 'false' ;

nil literal = 'nil' ;

character literal = ' ', ( character | escape sequence ), ' ' ;

escape sequence = '\\ ' | '\r' | '\n' | '\t' | '\f' | '\e' | '\a'
                  | '\x', 2 * hexadecimal digit
                  | '\u', 4 * hexadecimal digit
                  | '\c', character
                  | '\', 3 * octal digit
                  | '\"' | '\'' ;

text literal = '"', { '\ ' | character | escape sequence }, '"' ;

quote literal = '<', identifier, '>' ;

single line comment = '--', { character – newline }, newline ;

multiple line comment = '/*', { character }, '*/' ;
```

The escape sequences given above are to be interpreted as follows:



Sequence	Interpretation
'\\'	U+005C (backslash character)
'\r'	U+000D (return character)
'\n'	U+000A (newline character)
'\t'	U+0009 (tab character)
'\f'	U+000C (formfeed character)
'\e'	U+001B (escape character)
'\a'	U+0007 (alarm (bell))
'\x' hexadecimal digit, hexadecimal digit	U+00xy (hex representation of character (e.g. \x41 is 'A'))
'\u', 4 * hexadecimal digit	U+abcd (hex representation of character (e.g. \u0041 is 'A'))
'\c', character	U+00nn (control character) (e.g. \cA $\equiv$ \x01)
'\'', 3 * octal digit	U+00nn (octal representation of character)
'\"'	U+0022 (double quote)
'\''	U+0027 (apostrophe)

Table B.2: Escape sequences



# Appendix 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**,  $'| ->'$ ,  $'\dots'$ , 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	<b>comp</b>
2	<b>iterate</b>

## 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
           | arithmetic infix operator
           | set infix operator
           | sequence infix operator
           | map infix operator ;
```



## APPENDIX C. OPERATOR PRECEDENCE

arithmetic prefix operator = '+' | '-' | 'abs' | 'floor' ;

set prefix operator = 'card' | 'power' | 'dunion' | 'dinter' ;

sequence prefix operator = 'hd' | 'tl' | 'len'  
| 'inds' | 'elems' | 'conc' | 'reverse' ;

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	+ -	<b>union</b> \	<b>munion</b> ++	^
2	* / <b>rem</b> <b>mod</b> <b>div</b>	<b>inter</b>		
3			<b>inverse</b>	
4			<: <-:	
5			:> :->	
6	(unary) + (unary) - <b>abs</b> <b>floor</b>	<b>card</b> <b>power</b> <b>dinter</b> <b>dunion</b>	<b>dom</b> <b>rng</b> <b>merge</b>	<b>len</b> <b>elems</b> <b>hd tl</b> <b>conc</b> <b>inds</b> <b>reverse</b>

Table C.1: Operator precedence



## 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	<=	<
	>=	>
	=	<>
	<b>subset</b>	<b>psubset</b>
	<b>in set</b>	<b>not in set</b>

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	<b>or</b>
4	<b>and</b>
5	<b>not</b>



## 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<sup>1</sup>.

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).
5. Unary type operators: `seq of`, `seq1 of`, `set of`, `set1 of`.

<sup>1</sup>Except the “map domain restrict to” and the “map domain restrict by” operators which have a right grouping. This is not standard.



# Appendix D

## Differences between the Concrete Syntaxes

When VDM was originally developed a mathematical syntax was used and this have also been retained in the ISO/VDM-SL standard. However, most VDM tools today mainly use an ASCII syntax. Below is a list of the symbols which are different in the mathematical syntax and the ASCII syntax:

Mathematical syntax	ASCII syntax
$\cdot$	<code>&amp;</code>
$\times$	<code>*</code>
$\leq$	<code>&lt;=</code>
$\geq$	<code>&gt;=</code>
$\neq$	<code>&lt;&gt;</code>
$\xrightarrow{o}$	<code>==&gt;</code>
$\rightarrow$	<code>-&gt;</code>
$\Rightarrow$	<code>=&gt;</code>
$\Leftrightarrow$	<code>&lt;=&gt;</code>
$\mapsto$	<code>  -&gt;</code>
$\triangle$	<code>==</code>
$\uparrow$	<code>**</code>
$\dagger$	<code>++</code>
$\sqcup$	<code><b>munion</b></code>
$\triangleleft$	<code>&lt;:</code>
$\triangleright$	<code>:&gt;</code>
$\triangleleft$	<code>&lt;-:</code>
$\triangleright$	<code>:-&gt;</code>
$\cup$	<code><b>psubset</b></code>
$\cup$	<code><b>subset</b></code>
$\supset$	<code><b>dinter</b></code>
$\supset$	<code><b>dunion</b></code>



Mathematical syntax	ASCII syntax
$\mathcal{F}$	<b>power</b>
$\dots\text{-set}$	<b>set of ...</b>
$\dots\text{-set}_1$	<b>set1 of ...</b>
$\dots^*$	<b>seq of ...</b>
$\dots^+$	<b>seq1 of ...</b>
$\dots \xrightarrow{m} \dots$	<b>map ... to ...</b>
$\dots \xleftrightarrow{m} \dots$	<b>inmap ... to ...</b>
$\mu$	<b>mu</b>
$\mathbb{B}$	<b>bool</b>
$\mathbb{N}$	<b>nat</b>
$\mathbb{N}_1$	<b>nat1</b>
$\mathbb{Z}$	<b>int</b>
$\mathbb{R}$	<b>real</b>
$\neg$	<b>not</b>
$\cap$	<b>inter</b>
$\cup$	<b>union</b>
$\in$	<b>in set</b>
$\notin$	<b>not in set</b>
$\wedge$	<b>and</b>
$\vee$	<b>or</b>
$\forall$	<b>forall</b>
$\exists$	<b>exists</b>
$\exists!$	<b>exists1</b>
$\lambda$	<b>lambda</b>
$\iota$	<b>iota</b>
$\dots^{-1}$	<b>inverse ...</b>