

INTERNATIONAL STANDARD

ISO/IEC
10514-1

First edition
1996-06-01

Information technology — Programming languages —

Part 1: Modula-2, Base Language

*Technologies de l'information — Langages de programmation —
Partie 1: Modula-2, langage de base*



Reference number
ISO/IEC 10514-1:1996(E)

AVA Documents
Document Sharing Hub

Contents	Page
Foreword	xiv
Introduction	xv
Section 1 : Scope	1
1.1 Goals	1
1.2 Specifications included in this part of ISO/IEC 10514	1
1.3 Specifications not within the scope of this part of ISO/IEC 10514	1
Section 2 : Normative References	3
Section 3 : Definitions, Structure, and Conventions	4
3.1 Definitions	4
3.2 Structure of the Formal Definition	6
3.3 Conventions	8
3.3.1 Informative text	8
3.3.2 Typographical conventions	8
3.3.3 References to pervasive identifiers	9
3.3.4 Interpretation of non-VDM-SL notations	9
3.3.5 References to formal parameter names	9
Section 4 : Requirements for Implementations	10
4.1 Translation	10
4.2 Source Code Representation	10
4.3 Ordering of Declarations	11
4.4 Predefined Entities	11
4.5 Library Modules	11
4.6 Errors	12
4.7 Exceptions	12
4.8 Implementation-dependencies	13
4.9 Documentation	13
4.10 Statement of Compliance	14
4.11 Minimum requirements	14

Section 5 : The Lexis	16
5.1 Source Code Structure	16
5.2 Word tokens	16
5.2.1 Identifiers	16
5.2.2 Portable Identifiers	16
5.2.3 Full Identifiers	16
5.2.4 Pervasive Identifiers	16
5.2.5 Keywords	17
5.3 Symbols and Operators	17
5.3.1 Required Symbols	17
5.3.2 Symbols with alternatives	18
5.3.3 Operators	18
5.4 Constant Literals	20
5.4.1 Whole Number Literals	20
5.4.2 Real Literals	20
5.4.3 String Literals	20
5.5 Separators	21
5.5.1 White Space	21
5.5.2 Comments	21
5.5.3 Source Code Directives	21
5.6 Letters and National Characters	21
Section 6 : The Language	23
6.1 Programs, Program Modules, and Separate Modules	23
6.1.1 Programs and Compilation Modules	23
6.1.2 Program Modules	27
6.1.3 Definition Modules	28
6.1.4 Implementation Modules	30
6.1.4.1 Sourced Implementation Modules	31
6.1.4.2 Abstract Implementation Modules	33
6.1.5 Program Module and Separate Module Consistency	34
6.1.5.1 Definition Module and Implementation Module Consistency	34
6.1.5.2 Module Compilation Order	36
6.1.5.3 Module Dependencies	37
6.1.6 Module Initialization Order	38
6.1.7 Program Termination	40
6.1.8 Import Lists	41
6.1.8.1 Import List	43
6.1.8.2 Simple Imports	43
6.1.8.3 Unqualified Import	44
6.1.8.4 Import Consistency in Definition Modules	45
6.1.8.5 Import Consistency in Module Blocks	46
6.1.9 Export Lists	46
6.1.9.1 Unqualified Exports	46
6.1.9.2 Qualified Exports	47
6.1.10 Implicit Import and Export	48
6.1.11 Protected Modules	50
6.2 Definitions and Declarations	52
6.2.1 Identifiers	52
6.2.2 Qualified Identifiers	52
6.2.3 Definitions	54
6.2.4 Type Definitions	56
6.2.5 Procedure Headings	57
6.2.5.1 Proper Procedure Headings	58
6.2.5.2 Function Procedure Headings	60
6.2.5.3 Formal Parameters	61
6.2.6 Declarations	63

6.2.7	Ordering of Procedure and Module Declarations	66
6.2.8	Constant Declarations	67
6.2.9	Type Declarations	68
6.2.10	Variable Declarations	69
6.2.11	Procedure Declarations	71
6.2.11.1	Proper Procedure Declarations	73
6.2.11.2	Function Procedure Declarations	76
6.2.12	Local Module Declarations	79
6.2.13	Auxiliary Formal Model Definitions	82
6.2.13.1	Identifiers Defined in Definitions	82
6.2.13.2	Identifiers Declared in Declarations	82
6.2.13.3	Identifiers Defined in New Types	83
6.2.13.4	Qualified Identifiers Used in Declarations	84
6.2.13.5	Qualified Identifiers used in Expressions	84
6.2.13.6	Qualified Identifiers Used in Types	87
6.2.13.7	Qualified Identifiers Used in Parameters	88
6.2.13.8	Shielded Environments and Shielded Identifiers	89
6.3	Data Types	91
6.3.1	Type Denoters and Ordinal Type Denoters	91
6.3.2	Type Identifiers	92
6.3.3	New Types	93
6.3.4	Enumeration Types	94
6.3.5	Subrange Types	95
6.3.6	Set Types	97
6.3.7	Packedset Types	98
6.3.8	Pointer Types	99
6.3.9	Procedure Types	100
6.3.9.1	Proper Procedure Types	101
6.3.9.2	Function Procedure Types	101
6.3.9.3	Formal Parameter Type Lists	102
6.3.9.4	Variable Formal Types	103
6.3.9.5	Value Formal Types	104
6.3.10	Formal Types	104
6.3.11	Array Types	106
6.3.12	Record Types	107
6.3.12.1	Fixed Fields	109
6.3.12.2	Variant Fields	109
6.3.13	Storage Allocation	114
6.3.13.1	Allocating Storage for Constructed Types	114
6.3.13.2	Allocating Storage for Elementary Types	115
6.3.13.3	Allocating Storage for Structured (Composite) Types	116
6.3.13.4	Allocating Storage for System Storage Types	120
6.4	Expression and Assignment Compatibility	121
6.4.1	Expression Compatibility	121
6.4.2	Assignment Compatibility	122
6.5	Blocks	126
6.5.1	Proper Procedure Blocks	126
6.5.2	Function Procedure Blocks	129
6.5.3	Module Blocks	131
6.5.4	Static Module Initialization	132
6.5.5	Static Module Finalization	135
6.5.6	Dynamic Module Initialization	136
6.5.7	Dynamic Module Finalization	138
6.5.8	Block Bodies and Exception Handling	140
6.5.8.1	Non-result Bodies	142
6.5.8.2	Result Bodies	144
6.6	Statements	146

6.6.1	Statement Sequences	147
6.6.2	Empty Statements	148
6.6.3	Assignment Statements	148
6.6.4	Procedure Calls	152
6.6.5	Return Statements	153
6.6.5.1	Simple Return Statements	154
6.6.5.2	Function Return Statements	154
6.6.6	Retry Statements	155
6.6.7	With Statements	156
6.6.8	If Statements	157
6.6.9	Case Statements	159
6.6.9.1	Case Alternatives	160
6.6.10	While Statements	161
6.6.11	Repeat Statements	162
6.6.12	Loop Statements	163
6.6.13	Exit Statements	164
6.6.14	For Statements	164
6.6.15	Well-formed Control Variables	168
6.6.15.1	Threatening	169
6.6.16	Auxiliary Functions	176
6.6.16.1	Well-formed Return Statements	176
6.6.16.2	Well-formed Retry Statements	177
6.6.16.3	Well-formed Exit Statements	177
6.6.16.4	Statement Classes of Statement Sequences	179
6.7	Variable Designators	182
6.7.1	Entire Designators	182
6.7.2	Indexed Designators	184
6.7.3	Selected Designators	185
6.7.4	Dereferenced Designators	187
6.8	Expressions	189
6.8.1	Ordinal Expressions	190
6.8.2	Infix Expressions and Operations	190
6.8.2.1	Complex Number Infix Operations	193
6.8.2.2	Real Number Infix Operations	196
6.8.2.3	Whole Number Infix Operations	198
6.8.2.4	Boolean Infix Operations	202
6.8.2.5	Set Infix Operations	203
6.8.2.6	String Literal Infix Operations	205
6.8.2.7	Relational Operations	206
6.8.3	Prefix Expressions	219
6.8.3.1	Arithmetic Prefix Operations	220
6.8.3.2	Boolean Prefix Operations	221
6.8.4	Value Designators	222
6.8.4.1	Entire Values	223
6.8.4.2	Indexed Values	225
6.8.4.3	Selected Values	226
6.8.4.4	Dereferenced Values	227
6.8.5	Function Calls	229
6.8.6	Value Constructors	230
6.8.6.1	Array Constructors	231
6.8.6.2	Record Constructors	236
6.8.6.3	Set Constructors	241
6.8.7	Constant Literals	247
6.8.7.1	Whole Number Literals	248
6.8.7.2	Real Literals	249
6.8.7.3	String Literals	250
6.8.8	Constant Expressions	253

6.8.8.1	Constant Expression Evaluation	257
6.9	Parameter Compatibility and Argument Binding	258
6.9.1	Actual Parameters	258
6.9.1.1	Variable Actual Parameters	259
6.9.1.2	Expression Actual Parameters	260
6.9.1.3	Type Parameters	260
6.9.2	Parameter Matching	261
6.9.3	Parameter Compatibility	261
6.9.3.1	Value Parameter Compatibility	262
6.9.3.2	Variable Parameter Compatibility	263
6.9.3.3	System Storage Parameter Compatibility	264
6.9.4	Argument Binding	265
6.9.4.1	Argument Binding to Value Parameters	266
6.9.4.2	Argument Binding to Variable Parameters	267
6.10	Predefined Types, Standard Procedures and Standard Functions	270
6.10.1	The Pervasive Environment	270
6.10.2	Predefined Types	271
6.10.2.1	Predefined Whole Number Types	271
6.10.2.2	Predefined Real Number Types	272
6.10.2.3	Predefined Complex Number Types	273
6.10.2.4	The Boolean Type	274
6.10.2.5	The Character Type	274
6.10.2.6	Values of the Basic Types	276
6.10.2.7	The String Literal Type	278
6.10.2.8	The Nil Type	279
6.10.2.9	The Bitset Type	279
6.10.2.10	The Proc Type	279
6.10.2.11	The Protection Type	280
6.10.3	Standard Procedures	280
6.10.3.1	Standard Procedure Designators	280
6.10.3.2	The Procedure DEC	281
6.10.3.3	The Procedure DISPOSE	283
6.10.3.4	The Procedure EXCL	284
6.10.3.5	The Procedure HALT	285
6.10.3.6	The Procedure INC	286
6.10.3.7	The Procedure INCL	286
6.10.3.8	The Procedure NEW	287
6.10.4	Standard Functions	288
6.10.4.1	Standard Function Designators	288
6.10.4.2	The Function ABS	290
6.10.4.3	The Function CAP	291
6.10.4.4	The Function CHR	292
6.10.4.5	The Function CMPLX	292
6.10.4.6	The Function FLOAT	293
6.10.4.7	The Function HIGH	294
6.10.4.8	The Function IM	295
6.10.4.9	The Function INT	296
6.10.4.10	The Function LENGTH	297
6.10.4.11	The Function LFLOAT	297
6.10.4.12	The Function MAX	298
6.10.4.13	The Function MIN	299
6.10.4.14	The Function ODD	300
6.10.4.15	The Function ORD	301
6.10.4.16	The Function RE	301
6.10.4.17	The Function SIZE	302
6.10.4.18	The Function TRUNC	303
6.10.4.19	The Function VAL	304

6.10.5	Auxiliary Definitions	306
6.11	The Environment and Auxiliary Formal Definitions	307
6.11.1	Environments	307
6.11.1.1	Constant Values	308
6.11.1.2	Types	308
6.11.1.3	Structures	309
6.11.1.4	Expression Types	311
6.11.1.5	Types for Declared Procedures	311
6.11.1.6	Types for Predefined Procedures	311
6.11.1.7	Module Environments	312
6.11.1.8	Variables	312
6.11.1.9	Protection Domains	313
6.11.1.10	Local Continuations	313
6.11.2	Operations on the Environment	313
6.11.2.1	Functions Accessing the Constants Component of the Environment	313
6.11.2.2	Functions Accessing the Types Component of the Environment	314
6.11.2.3	Functions Accessing the Structures Component of the Environment	315
6.11.2.4	Functions Accessing the Variables Component of the Environment	315
6.11.2.5	Functions Accessing the Procedures Component of the Environment	316
6.11.2.6	Functions Accessing the Modules Component of the Environment	317
6.11.2.7	Functions Accessing the Procedure Level Component of the Environment	317
6.11.2.8	Functions Accessing the Continuations Component of the Environment	318
6.11.2.9	Functions Accessing the Protection Components of the Environment	318
6.11.2.10	Functions Accessing the Environment	319
6.11.2.11	Operations to Update Environments	319
6.11.3	Types	320
6.11.3.1	New Type Names	320
6.11.3.2	Structured (Composite) Types	321
6.11.3.3	Elementary Types	321
6.11.3.4	Scalar Types	322
6.11.3.5	Ordinal Types	322
6.11.3.6	Whole Number Types	323
6.11.3.7	Number Types	324
6.11.3.8	String Types	324
6.11.3.9	System Storage Types	324
6.11.4	Unit Types	325
6.11.4.1	Range Type	325
6.11.4.2	Host Type	325
6.11.4.3	Base Type	325
6.11.4.4	Component Type	325
6.11.4.5	Index Type	326
6.11.4.6	Bound Type	326
6.11.5	Other Type Functions	326
6.11.5.1	Return Types	326
6.11.5.2	Formal Parameter Types	326
6.11.5.3	Auxiliary Functions for Records	326
6.11.5.4	Auxiliary Functions for Arrays	328
6.11.5.5	Auxiliary Functions for Open Arrays	328
6.11.5.6	Auxiliary Functions for Opaque Types	328
6.11.6	Values Associated with a Type	328
6.12	The Storage Model and Auxiliary Formal Definitions	331
6.12.1	The Program State	331
6.12.2	The External Program State	331
6.12.2.1	Operations to Initialize and Terminate the Program State	331
6.12.3	The Store	332
6.12.3.1	Storage	332
6.12.3.2	Values	333

6.12.4	Procedure Denotations	334
6.12.5	Termination	334
6.12.6	Exceptions	334
6.12.7	Interrupt Handlers	335
6.12.8	Operations on Storage	335
6.12.8.1	Operations Accessing Storage	335
6.12.8.2	Storage Aliasing	336
6.12.8.3	Operations to Allocate and Deallocate Storage	337
6.12.8.4	Operations to Allocate and Deallocate Storage Locations	338
6.12.8.5	Access Coroutine Storage	339
6.12.9	Accessing the Procedure Denotation Component of the State	339
6.12.10	Program Initialization and Program Termination	340
6.12.10.1	Program Initialization	340
6.12.10.2	Program Termination	340
6.12.10.3	Program Halting	341
6.12.11	Exception Handling	341
6.12.11.1	Allocation of Exception Sources	342
6.12.12	Interrupt Handlers	343
6.12.13	Accessing the Protection Component of the State	344
6.12.14	Coroutine and Continuations	344
6.12.14.1	The Coroutine Environment	344
6.12.14.2	The Continuations	344
6.12.14.3	Operations on Continuations	345

Section 7 : System Modules 347

7.1	The Module SYSTEM	348
7.1.1	The Interface to SYSTEM	348
7.1.2	The Constants of SYSTEM	351
7.1.3	The Types of SYSTEM	352
7.1.3.1	System Storage Types	352
7.1.3.2	The Address Type	353
7.1.4	The Functions of SYSTEM	353
7.1.5	Address Arithmetic Functions	354
7.1.5.1	The Function ADDADR	355
7.1.5.2	The Function SUBADR	356
7.1.5.3	The Function DIFADR	357
7.1.5.4	Properties of the address arithmetic functions	358
7.1.6	Address Construction and Enquiry Functions	359
7.1.6.1	The Function MAKEADR	359
7.1.6.2	The Function ADR	359
7.1.7	Packedset Functions	360
7.1.7.1	Packedset mapping	361
7.1.7.2	The Function ROTATE	361
7.1.7.3	The Function SHIFT	362
7.1.8	The Function CAST	364
7.1.9	The Function TSIZE	365
7.2	The Module COROUTINES	368
7.2.1	The Interface to COROUTINES	368
7.2.2	The Types of COROUTINES	369
7.2.3	The Procedures of COROUTINES	370
7.2.3.1	The Procedure ATTACH	371
7.2.3.2	The Procedure DETACH	371
7.2.3.3	The Procedure IOTRANSFER	372
7.2.3.4	The Procedure LISTEN	373
7.2.3.5	The Procedure NEWCOROUTINE	373
7.2.3.6	The Procedure TRANSFER	376
7.2.4	The Functions of COROUTINES	376

7.2.4.1	The Function HANDLER	377
7.2.4.2	The Function IsATTACHED	378
7.2.4.3	The Function CURRENT	379
7.2.4.4	The Function PROT	379
7.3	The Module EXCEPTIONS	380
7.3.1	The Interface to EXCEPTIONS	380
7.3.2	The Types of EXCEPTIONS	381
7.3.3	The Procedures of EXCEPTIONS	381
7.3.3.1	The Procedure AllocateSource	382
7.3.3.2	The Procedure RAISE	383
7.3.3.3	The Procedure GetMessage	383
7.3.4	The Functions of EXCEPTIONS	384
7.3.4.1	The Function CurrentNumber	385
7.3.4.2	The Function IsCurrentSource	386
7.3.4.3	The Function IsExceptionalExecution	387
7.3.4.4	The Use of EXCEPTIONS	387
7.4	The Module TERMINATION	389
7.4.1	The Interface to TERMINATION	389
7.4.2	The Functions of TERMINATION	389
7.4.2.1	The Function IsTerminating	390
7.4.2.2	The Function HasHalted	391
7.5	The Module M2EXCEPTION	392
7.5.1	The Interface to M2EXCEPTION	392
7.5.2	The Types of M2EXCEPTION	393
7.5.3	The Functions of M2EXCEPTION	393
7.5.3.1	The Function M2Exception	394
7.5.3.2	The Function IsM2Exception	395
7.5.4	Language Exceptions	395
7.5.4.1	Exceptions whose detection is required	396
7.5.4.2	Exceptions whose detection is not required	397
7.5.5	Messages Associated with Language Exceptions	397
7.5.6	Aggregation and Raising of Language Exceptions	398
Section 8	Required Library Modules	399
8.1	The Module Storage	399
8.1.1	The Definition Module of Storage	399
8.1.2	The Procedures of Storage	400
8.1.3	Exceptions Raised by Storage	402
8.1.4	The State for Storage	402
8.1.5	The Initialization of Storage	403
8.1.6	Auxiliary Operations	403
8.2	The Modules LowReal and LowLong	405
8.2.1	The Definition Module of LowReal	405
8.2.2	The Definition Module of LowLong	406
8.2.3	The Constants of LowReal and LowLong	407
8.2.4	The Procedures of LowReal and LowLong	409
8.2.5	Exceptions Raised by LowReal and LowLong	411
8.3	The Module CharClass	412
8.3.1	The Definition Module of CharClass	412
8.3.2	The Procedures of CharClass	412
Section 9	Standard Library Modules	415
9.1	The Mathematical Libraries	415
9.1.1	The Definition Module of RealMath	415
9.1.2	The Definition Module of LongMath	416
9.1.3	The Constants of RealMath and LongMath	417
9.1.4	The Procedures of RealMath and LongMath	417

9.1.5	Exceptions Raised by RealMath and LongMath	420
9.1.6	The State for RealMath and LongMath	421
9.1.7	The Initialization of RealMath and LongMath	421
9.1.8	The Definition Module of ComplexMath	421
9.1.9	The Definition Module of LongComplexMath	422
9.1.10	The Constants of ComplexMath and LongComplexMath	423
9.1.11	The Procedures of ComplexMath and LongComplexMath	423
9.1.12	Exceptions Raised by ComplexMath and LongComplexMath	428
9.1.13	The State for ComplexMath and LongComplexMath	429
9.1.14	The Initialization of ComplexMath and LongComplexMath	429
9.1.15	Auxiliary Functions	429
9.2	The Input/Output Library	430
9.2.1	Standard and Default Channels	431
9.2.1.1	The Module StdChans	431
9.2.2	Reading and Writing of Data using Specified Channels	434
9.2.2.1	The Module TextIO	435
9.2.2.2	The Module WholeIO	439
9.2.2.3	The Modules RealIO and LongIO	442
9.2.2.4	The Module RawIO	447
9.2.2.5	The Module IOConsts	448
9.2.2.6	The Module IOResult	449
9.2.2.7	Auxiliary Formal Model Definitions	450
9.2.3	Reading and Writing of Data using Default Channels	455
9.2.3.1	The Module STextIO	455
9.2.3.2	The Module SWholeIO	456
9.2.3.3	The Modules SRealIO and SLongIO	457
9.2.3.4	The Module SRawIO	459
9.2.3.5	The Module SIOResult	459
9.2.4	The Interface for Device-Independent Channel Operations	461
9.2.4.1	The Module IOChan	461
9.2.5	Obtaining Channels from Device Modules	472
9.2.5.1	The Module ChanConsts	472
9.2.5.2	The Module StreamFile	475
9.2.5.3	The Module SeqFile	483
9.2.5.4	The Module RndFile	497
9.2.5.5	The Module TermFile	511
9.2.5.6	The Module ProgramArgs	522
9.2.5.7	Implementation-defined Functions used in the Device Modules	529
9.2.5.8	Auxiliary Formal Model Definitions	529
9.2.6	The Interface to Channels for New Device Modules	531
9.2.6.1	The Module IOLink	531
9.2.7	The State for the Input/Output Library	536
9.3	Concurrent Programming	544
9.3.1	The Module Processes	544
9.3.1.1	The Definition Module of Processes	546
9.3.1.2	The Procedures of Processes	548
9.3.1.3	Exceptions Raised by Processes	554
9.3.1.4	The State for Processes	555
9.3.1.5	The Initialization of Processes	555
9.3.1.6	Auxiliary Formal Model Definitions	556
9.3.2	The Module Semaphores	559
9.3.2.1	The Definition Module of Semaphores	559
9.3.2.2	The Procedures of Semaphores	559
9.3.2.3	Exceptions Raised by Semaphores	562
9.3.2.4	The State for Semaphores	562
9.3.2.5	The Initialization of Semaphores	563
9.3.2.6	Auxiliary Formal Model Definitions	563

9.4	The Module Strings	564
9.4.1	The Definition Module of Strings	564
9.4.2	The Procedures of Strings	567
9.4.3	Auxiliary Formal Model Definitions	579
9.5	The String Conversions Library	581
9.5.1	Common Data Types	581
9.5.1.1	The Module ConvTypes	581
9.5.2	High-Level String Conversion Modules	582
9.5.2.1	The Module WholeStr	582
9.5.2.2	The Modules RealStr and LongStr	586
9.5.3	Low-Level String Conversion Modules	592
9.5.3.1	The Module WholeConv	592
9.5.3.2	The Modules RealConv and LongConv	598
9.5.4	Auxiliary Definitions	605
9.6	The Module SysClock	607
9.6.1	The Definition Module of SysClock	607
9.6.2	The Types of SysClock	608
9.6.3	The Procedures of SysClock	608
9.6.4	The State for SysClock	609
Section 10:	Auxiliary Formal Model Definitions	611
10.1	Extensions to VDM-SL	611
10.1.1	Replacement Definitions for Combinators	611
10.1.2	The McCarthy Conditional Expression	612
10.2	VDM-SL Functions	614
10.2.1	Set functions	614
10.2.2	Sequence functions	614
10.2.3	Map functions	615
10.2.4	Numerical functions	615

Annexes

Annex A	Minimum Limit Specifications	616
A.1	The Limit-specification Generators	616
A.1.1	Length of Identifiers	616
A.1.2	Length of String Literals	617
A.1.3	Size of Single Dimension Arrays	617
A.1.4	Number of Dimensions of Multidimension Arrays	618
A.1.5	Depth of Nested Records	619
A.1.6	Number of Sequential Local Modules	620
A.1.7	Depth of Nested Local Modules	620
A.1.8	Number of Enumeration Literals	621
A.1.9	Number of Sequential Procedure Declarations	622
A.1.10	Depth of Nested Procedure Declarations	623
A.1.11	Number of Coroutines	624
A.1.12	Number of Parameters of a Procedure	625
A.1.13	Depth of Nested IF-THEN-ELSEs in THEN-branch	626
A.1.14	Depth of Nested IF-THEN-ELSEs in Both Branches	627
A.1.15	Depth of Nested LOOP Statements	628
A.1.16	Depth of Nested WHILE Statements	629
A.1.17	Depth of Nested REPEAT Statements	630
A.1.18	Depth of Nested FOR Statements	631
A.1.19	Depth of Nested WITH Statements	632
A.1.20	Number of CASE Alternatives with ELSE part	633
A.1.21	Depth of Nested Parentheses in Expressions	634
A.1.22	Depth of Recursion	635
A.1.23	Number of Imported Modules	635

Annex B The Specification of Library Modules	637
B.1 Components of a Library Module Specification	637
B.2 Interpretation of a Library Module Specification	637
Annex C Collected Modula-2 Concrete Syntax	639
C.1 Programs, Program Modules, and Separate Modules	639
C.1.1 Programs and Compilation Modules	639
C.1.2 Program Modules	639
C.1.3 Definition Modules	639
C.1.4 Implementation Modules	639
C.1.4.1 Sourced Implementation Modules	639
C.1.5 Import Lists	639
C.1.5.1 Import List	639
C.1.5.2 Simple Imports	639
C.1.5.3 Unqualified Import	639
C.1.6 Export Lists	640
C.1.6.1 Unqualified Exports	640
C.1.6.2 Qualified Exports	640
C.1.7 Protected Modules	640
C.2 Definitions and Declarations	640
C.2.1 Qualified Identifiers	640
C.2.2 Definitions	640
C.2.3 Type Definitions	640
C.2.4 Procedure Headings	640
C.2.4.1 Proper Procedure Headings	640
C.2.4.2 Function Procedure Headings	641
C.2.4.3 Formal Parameters	641
C.2.5 Declarations	641
C.2.6 Constant Declarations	641
C.2.7 Type Declarations	641
C.2.8 Variable Declarations	641
C.2.9 Procedure Declarations	641
C.2.9.1 Proper Procedure Declarations	642
C.2.9.2 Function Procedure Declarations	642
C.2.10 Local Module Declarations	642
C.3 Data Types	642
C.3.1 Type Denoters and Ordinal Type Denoters	642
C.3.2 Type Identifiers	642
C.3.3 New Types	642
C.3.4 Enumeration Types	642
C.3.5 Subrange Types	642
C.3.6 Set Types	643
C.3.7 Packedset Types	643
C.3.8 Pointer Types	643
C.3.9 Procedure Types	643
C.3.9.1 Proper Procedure Types	643
C.3.9.2 Function Procedure Types	643
C.3.9.3 Formal Parameter Type Lists	643
C.3.9.4 Variable Formal Types	643
C.3.9.5 Value Formal Types	643
C.3.10 Formal Types	643
C.3.11 Array Types	644
C.3.12 Record Types	644
C.3.12.1 Fixed Fields	644
C.3.12.2 Variant Fields	644
C.4 Blocks	644
C.4.1 Proper Procedure Blocks	644

C.4.2	Function Procedure Blocks	644
C.4.3	Module Blocks	645
C.4.4	Block Bodies and Exception Handling	645
C.5	Statements	645
C.5.1	Statement Sequences	645
C.5.2	Empty Statements	645
C.5.3	Assignment Statements	645
C.5.4	Procedure Calls	645
C.5.5	Return Statements	645
C.5.5.1	Simple Return Statements	645
C.5.5.2	Function Return Statements	646
C.5.6	Retry Statements	646
C.5.7	With Statements	646
C.5.8	If Statements	646
C.5.9	Case Statements	646
C.5.9.1	Case Alternatives	646
C.5.10	While Statements	646
C.5.11	Repeat Statements	646
C.5.12	Loop Statements	647
C.5.13	Exit Statements	647
C.5.14	For Statements	647
C.6	Variable Designators	647
C.6.1	Entire Designators	647
C.6.2	Indexed Designators	647
C.6.3	Selected Designators	647
C.6.4	Dereferenced Designators	647
C.7	Expressions	648
C.7.1	Ordinal Expressions	648
C.7.2	Infix Expressions and Operations	648
C.7.3	Value Designators	648
C.7.3.1	Entire Values	648
C.7.3.2	Indexed Values	648
C.7.3.3	Selected Values	648
C.7.3.4	Dereferenced Values	649
C.7.4	Function Calls	649
C.7.5	Value Constructors	649
C.7.5.1	Array Constructors	649
C.7.5.2	Record Constructors	649
C.7.5.3	Set Constructors	649
C.7.6	Constant Literals	650
C.7.7	Constant Expressions	650
C.8	Parameter Compatibility and Argument Binding	650
C.8.1	Actual Parameters	650
C.8.1.1	Type Parameters	650
C.9	Lexical components	650
Annex D	Collected Modula-2 Abstract Syntax	652
Annex E	Modula-2 Glossary	660
Annex F	Participating Individuals and Organisations	674
Index		676

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 10514-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 10514 consists of the following parts, under the general title *Information technology — Programming languages*:

- *Part 1: Modula-2, Base Language*
- *Part 2: Object-oriented Modula-2*

Annexes A and B form an integral part of this part of ISO/IEC 10514. Annexes C to F are for information only.

Introduction

This part of ISO/IEC 10514, Modula-2 (Base language), provides a specification of the form and meaning of Modula-2 programs, and by reference to that specification lays down requirements for Modula-2 implementations.

Modula-2 is a general-purpose programming language, characterized by modules that provide data and procedure encapsulation and hiding. It provides facilities for concurrent programming based on coroutines, and includes facilities for low-level programming. Modula-2 is, therefore, suitable for systems programming. Facilities such as input/output, mathematical operations and string processing are made available through library modules.

Modula-2 is a descendent of Modula whose ancestor is Pascal. All three languages were devised by Professor Niklaus Wirth, ETH, Zürich, Switzerland. The first description of Modula-2 was published in 1980. The language became popular during the early 1980s and many implementations were developed. Because of different interpretations of various descriptions of Modula-2, significant differences in implementations appeared, and in 1984 a group of Modula-2 proponents formed a BSI panel to develop a standard. The BSI panel successfully proposed a new work item to ISO, and the ISO working group met for the first time in 1987. In 1989 the first draft of this standard was published, and in 1992 the second committee draft was published. The base documents for the formulation of this part of ISO/IEC 10514 were the various editions of *Programming in Modula-2* by Niklaus Wirth, published by Springer-Verlag.

This part of ISO/IEC 10514 contains a formal model of Modula-2; for the sake of generality, consistency, and precision, differences have been introduced between the model and the third edition of Professor Wirth's book *Programming in Modula-2* and these are identified either as 'clarifications' or 'changes'. During the standardization of Modula-2 it was recognized that users of the language were demanding extensions to the language devised by Professor Wirth and that implementors were responding by providing them. Several changes and extensions have, therefore, been made in order to prevent further divergence, and to ensure consistency and coherence.

This part of ISO/IEC 10514 uses a variety of notations to specify the requirements contained in it. A standard syntactic metalanguage is used for specifying lexical and syntactic structures. The Vienna Development Method-Specification Language (VDM-SL), which is based upon conventional mathematics, is used to define the semantics of the language in a mathematically rigorous manner. In addition, the semantics of the Modula-2 language, system modules and standard library are also specified using natural language, albeit in a form that is possibly ambiguous. It is intended that the corresponding VDM-SL resolves any such ambiguity. If the natural language requirements and those expressed in the VDM-SL conflict, then there is a mistake in this standard. In many cases, small details of the language semantics may only be resolved by reference to the VDM-SL; by this means the natural language description has been kept relatively simple. A number of typographic and stylistic conventions have been adopted in order to make the natural language description succinct.

The definitions of key terms used in this part of ISO/IEC 10514 are expressed in natural language, with references to VDM-SL where appropriate. The requirements for conforming implementations are given in natural language, with references to VDM-SL where appropriate.

The development of large parts of the formal specification of the semantics of Modula-2 in this part of ISO/IEC 10514 was sponsored by the U.K. Alvey Research Programme. The Alvey sponsored work was carried out at Leicester University Computer Studies Department in collaboration with BSI QA Software Engineering Department.

Information technology — Programming languages —

Part 1:

Modula-2, Base Language

Section 1 : Scope

1.1 Goals

The goals of this part of ISO/IEC 10514 are:

- to provide a rigorous definition of the language Modula-2 and its standard library by providing a mathematical model of both;
- to provide a resolution of differences among interpretations of other descriptions of Modula-2 and its standard library, while endeavouring to preserve investment in existing practice;
- to remove features thought to be redundant, inherently flawed, or inadequate;
- to specify new language and standard library facilities where a need is perceived to exist;
- to maintain the general principles of Modula-2 laid down by its inventor, while allowing for later modernization and standardization.

1.2 Specifications included in this part of ISO/IEC 10514

This part of ISO/IEC 10514 provides specifications for:

- required symbols for Modula-2 program representation, including comments, literals, and source code directives;
- the lexical structure, the syntactic structure and the semantics of Modula-2 programs, including programs that make use of system modules.
- the interface to and the semantics of standard Modula-2 library modules.
- those separate modules of the standard library that a conforming implementation is required to supply;
- violations of the rules for use of the language, system modules and standard library modules that a conforming implementation is required to detect;
- certain criteria for the size and complexity of programs that a conforming implementation must accept;
- further compliance requirements for implementations, including documentation requirements.

1.3 Specifications not within the scope of this part of ISO/IEC 10514

This part of ISO/IEC 10514 provides no specifications for:

- the underlying representation of predefined data types (except in the case of packedset types; see 7.1.7.1);

- the method by which implementations are invoked (including identification of the program module and associated definition and implementation modules);
- the method by which compilation modules are stored (including the correspondence between module names and system file names where files are used);
- the method by which implementations accept input (including the encoding of source text and including the number of compilation modules accepted for each invocation);
- performance aspects of implementations, and certain quality aspects not covered by 1.2;
- the effect of executing a program that uses extensions to the language, extensions to system modules or extensions to standard library modules, or that otherwise deviates from this part of ISO/IEC 10514;
- the effect of continuing execution of a program in which an exception has occurred and execution has continued without an exception being raised;
- the meaning of a program that relies on a definition of implementation-dependent values or implementation-dependent behaviour.

Section 2 : Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 10514. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 10514 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 10967-1:1994, *Information technology—Language independent arithmetic—Part 1: Integer and floating point arithmetic*.

ISO/IEC 13817-1:…¹⁾, *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language—Part 1: Base language*.

BS 6154:1981, *Method of defining Syntactic Metalanguage*.

IEC 559:1989, *Binary floating-point arithmetic for microprocessor systems*

¹⁾To be published.

Section 3 : Definitions, Structure, and Conventions

3.1 Definitions

For the purposes of this part of ISO/IEC 10514, the following definitions apply.

NOTE 1 — A glossary of terms used is also provided — see annex E.

3.1.1 source code: A fragment of text that is potentially part of the text representation of a component of a Modula-2 program.

3.1.2 separate module: A component of a Modula-2 program, that may be separately developed, translated, and provided, and which comprises an encapsulation of data and/or procedures that are available for use by clients of the module, together with any actions that are executed to initialize the module, and any actions that are executed to finalize the module.

NOTE 2 — The interface to a separate module is provided by a Modula-2 definition module so that other separate modules written in Modula-2 can use its facilities. The executable part of a separate module is specified in Modula-2 by an implementation module; however, an implementation of a separate module need not be derived from Modula-2 source code.

3.1.3 library: A collection of separate modules provided for general use.

3.1.4 library module: A separate module that is part of a library.

3.1.5 standard library: The library consisting of the separate modules specified in this part of ISO/IEC 10514: the required modules in section 8 and the non-required modules in section 9.

3.1.6 predefined: A term which is applied to an entity (module, procedure, type or constant), or an identifier, that is built-in to Modula-2.

3.1.7 pervasive identifier: An identifier of a predefined procedure, type, or constant that is itself predefined and whose scope covers the entirety of compilation modules, unless overridden by an identifier of the same spelling that is introduced in a definition or a declaration.

3.1.8 system module: A predefined module that is an encapsulation of predefined entities whose identifiers are only available by import from that system module. Unlike separate modules, system modules cannot be replaced.

NOTE 3 — System modules do not form part of the standard library.

3.1.9 computer system: The combination of hardware and, optionally, firmware and software (e.g. operating system) that enables the execution of software.

3.1.10 Modula-2 implementation: The means by which an executable program for a computer system is produced from the components of a Modula-2 program.

An implementation comprises at least the following:

- a translator;
- any necessary run-time support for the predefined entities such as system modules and standard procedures;

- required separate modules;
- any software needed to support or integrate the above.

NOTES

- 4 The term 'translator' is intended to encompass technologies both known (e.g. compilers and interpreters) and those yet to be developed.
- 5 A translator need not work from source code input since an implementation can choose to use non-text representations that are deriveable from source code. — see 4.1.
- 6 A translator can deal with access to predefined entities in a special way (e.g. 'calls' to system module 'procedures' may be translated as inline code).

3.1.11 standard library module implementation: The means by which a module of the standard library is provided independently of a Modula-2 implementation.

3.1.12 implementor: The producer of a Modula-2 implementation.

3.1.13 processor: The combination of computer system and implementation that permits the translation and execution of Modula-2 programs. A 'processor' may include two computer systems if cross-translation is involved; in this situation an implementation's translator is operated on a **host** computer system and generates code for execution on a **target** computer system.

NOTE 7 — The term 'processor' does not preclude implementations that involve more than one hardware processor.

3.1.14 abstract implementation module: The specification of the semantics of a separate module using the formal notation VDM-SL — see annex B.

3.1.15 sourced implementation module: The implementation of a separate module for which source code is provided and whose semantics is therefore determined by the semantics of that source code as specified by this part of ISO/IEC 10514.

3.1.16 standard mode: A mode of use of an implementation in which it complies with the requirements of this part of ISO/IEC 10514.

3.1.17 implementation-defined: A term applied to a value that is not defined by this part of ISO/IEC 10514, but that requires definition before the meaning of a program that relies upon that value can be determined.

NOTE 8 — For example, the minimum and maximum values of the **INTEGER** type are implementation-defined, and the value of a call of **SYSTEM.CAST** is implementation-defined.

This part of ISO/IEC 10514 requires conforming implementations to be accompanied by documentation which includes definitions of implementation-defined values for each processor — see 4.9.

3.1.18 implementation-dependent: A term applied to a value or to behaviour that is not defined by this part of ISO/IEC 10514, and that requires no definition since the meaning of a program that relies upon such a definition is not specified by the terms of this part of ISO/IEC 10514.

NOTE 9 — For example, the order of evaluation of operands in an expression is implementation-dependent; consequently, a program that relies on the order of evaluation of operands in an expression is not given meaning by this part of ISO/IEC 10514.

3.1.19 error: Source code is said to be in error or to contain an error if either the source code is not well-formed (i.e. it does not conform to lexical or concrete syntax rules) or if syntactically correct source code is not well-formed according to static semantic rules. For example, the syntactically correct source code fragment `a1 + a2` is in error if the entity designated by `a1` (or `a2`) is an array.

3.1.20 exception: A run-time event that is in violation of the dynamic semantic rules of this part of ISO/IEC 10514.

NOTE 10 — For example, an exception occurs if there is an attempt to dereference a pointer variable whose value is `NIL`.

An exception should be detected and raised by an implementation, in which case there will be a defined control flow of the program that is different from the normal flow of control.

3.1.21 exception raising: The invocation of exception handling, or exceptional termination, following the detection of an exception occurrence.

NOTES

- 11 This part of ISO/IEC 10514 requires that the occurrence of certain exceptions be detected and raised by an implementation in standard mode. The function *mandatory-detection* is used in the formal definition in these cases and *non-mandatory-detection* is used otherwise — see 4.7.
- 12 If an implementation does not detect and raise an exception that has occurred, the effect is implementation-dependent; consequently, this part of ISO/IEC 10514 does not give meaning to a program that relies on the detection of an exception if that exception is one for which detection is not required — see 1.3.

3.1.22 declare-before-use-in-declarations: Implementations which only require that entities used in declarations be declared before they are used are described as declare-before-use-in-declarations. (They are often called ‘multi-pass’ implementations.) Declare-before-use-in-declarations implementations can, for example, process programs that contain local modules which import from each other. Implementations are termed declare-before-use-in-declarations if they satisfy the first predicate in 6.2.7.

3.1.23 declare-before-use: Implementations which insist that all entities except pointer types and procedure types be declared before they are used are described as declare-before-use. (They are often imprecisely called ‘single-pass’ implementations.) Declare-before-use implementations cannot, for example, process programs that contain local modules which import from each other. Implementations are termed declare-before-use if they satisfy the second predicate in 6.2.7.

3.2 Structure of the Formal Definition

This part of ISO/IEC 10514 uses both formal notations and natural language to state its requirements. For the most part the form and meaning of Modula-2 programs and the system modules, and library modules they use, is expressed using a formal definition accompanied by natural language normative statements and notes. The semantics of Modula-2 programs and standard libraries is expressed formally using VDM-SL (ISO/IEC 13817) with some local extensions which are described in 10.1.

Specification of language constructs

The formal definition of Modula-2 language constructs is divided into the lexis and five main parts for which headings appear under each language construct: concrete syntax, abstract syntax, declaration semantics, static semantics, and dynamic semantics. Other parts of the definition appear under the headings properties and domains.

3.2.1 lexis: The set of rules defining the lexical elements of Modula-2 source code in terms of characters or character sequences.

The Lexis is formally expressed using the *Standard Syntactic Metalanguage* (BS 6154, 1981) and is given in section 5.

3.2.2 concrete syntax: The set of grammatical rules, or ‘productions’, defining the syntactic structure of the language in terms of lexical tokens.

The concrete syntax is formally expressed using the *Standard Syntactic Metalanguage* (BS 6154, 1981) and is given separately for each construct in the language in section 6.

Terms from the concrete syntax are used in the natural language normative statements. The tokens of the concrete syntax have apparently meaningful names, and some productions are present solely to allow such terms to be used (for example, the token *constant expression* is simply defined as an *expression*). The actual meaning of these terms is expressed by the static semantics.

NOTE 1 — The concrete syntax rules are collected in informative annex C.

3.2.3 abstract syntax: An abstract version of the underlying structure of the language in terms of a hierarchy of VDM-SL types. Translation between the representations required by the concrete and abstract syntax is implicit. In a few cases, where the abstract syntax is not directly related to the concrete syntax, annotations are added to explain any apparent divergence.

The abstract syntax is used only in the VDM-SL specification.

NOTE 2 — The abstract syntax rules are collected in informative annex D.

3.2.4 declaration semantics: The set of rules giving the meaning of definitions, declarations, and of imports and exports, for the purpose of expressing the static semantics of the language. The rules define the scope of identifiers and their meaning within that scope — see 6.2. In this part of ISO/IEC 10514, the visibility and meaning of an identifier at the point of its use is determined by the environment value that applies at that point — see 6.11.1. The VDM-SL functions expressing the declaration semantics construct environment values from given environment values.

3.2.5 static semantics: The VDM-SL static semantics is the set of rules, expressed in the form of truth-valued functions (i.e. the ‘well-formedness’ predicates) on components of the abstract syntax, that specify the legality of an abstractly represented Modula-2 program.

The corresponding natural language normative statements express constraints that are not formally imposed by the concrete syntax (for example, that where a component called a *constant expression* is expected, that component is an expression that can be evaluated without executing the program).

3.2.6 dynamic semantics: The set of rules giving the meaning of the elaboration of the declarations, and the execution of the statements, of a program that satisfies the static semantics. In VDM-SL, the dynamic semantics are expressed by operations which implicitly operate on an internal state — see 6.12.1.

3.2.7 properties: A set of equations or inequalities that specify requirements by giving relations that must apply between values of the VDM-SL types that are used in the formal specification.

3.2.8 domains: The VDM-SL types whose values are used to build the mathematical model which expresses the semantics of a Modula-2 program. The two main such types are used to model the environment and the state; the other VDM-SL types given under the heading of domains are component types of one or both of the environment or the state.

Specification of predefined entities

Identifiers of predefined types, constants, procedures, and modules may appear in the concrete and abstract representations of a program but these predefined entities do not have definitions or declarations that appear within a program. The declaration semantics for predefined entities gives the meaning of these identifiers for the purpose of expressing the static semantics governing their use.

The static and dynamic semantics that apply to the use of predefined types, constants, and modules, are given by the static and dynamic semantics for the language constructs that allow the use of such types, constants, and modules. Static and dynamic semantics for calls of predefined procedures are given explicitly for each such procedure.

In the case of the specification of system modules, a pseudo-definition module is given for informational purposes only.

Specification of separate modules

In the case of the specification of separate modules, definition modules are given to which the declaration semantics, static semantics, and dynamic semantics of definition modules are to be applied.

The dynamic semantics of calls of procedures exported by a separate module are given explicitly for each such procedure and form part of an abstract implementation module for the separate module. Rules for binding abstract implementation modules to the rest of the formal definition are given in normative annex B.

3.3 Conventions

The following conventions used in this part of ISO/IEC 10514 are to be interpreted as described below.

3.3.1 Informative text

At certain points in this part of ISO/IEC 10514, informative, rather than normative text appears. In particular, such text often appears in introductory paragraphs immediately after clause or sub-clause headings (before the specification of concrete syntax or static semantics), and within the confines of sections where it is demarcated as notes, examples, clarifications, and changes, as exemplified below.

NOTE — Notes are usually printed in a smaller font than normative text.

The pseudo-definition modules of system modules are to be regarded as informative text.

Where necessary, parts of the VDM-SL are explained by means of informative natural language comments labelled as annotations.

3.3.2 Typographical conventions

The following typographical conventions are observed.

- Lexical production rules and concrete syntax rules are printed in a sans-serif font. Meta-identifiers are printed in this same font, e.g. national character.
- VDM-SL functions, operations and values are printed in an italic font, e.g. *t-variable-designator*.
- Where a function or operation is used in a VDM-SL function or operation, the page number where the function or operation is defined is given on the right-hand side of the page.
- Fragments of Modula-2 or keywords etc. are printed in a monospaced font, e.g.

CONST PI = 3.1415926535897932384626433832795028841972;

— Some definition modules of standard library modules or pseudo-definition modules of system modules contain implementation-defined types and implementation-defined constant values. These are shown between angle brackets < and >.

3.3.3 References to pervasive identifiers

This part of ISO/IEC 10514 gives meaning to a program in which identifiers spelt in the same way as the pervasive identifiers are introduced in definitions or declarations of new entities, unrelated to the predefined entities. In such a program the meaning of a pervasive identifier is overridden in the scope of an identifier of the same spelling that is introduced in a definition or a declaration. This part of ISO/IEC 10514 refers throughout to pervasive identifiers in a monospaced font (e.g. `BOOLEAN`), *with the implication that their original, predefined, meaning is intended*. For example, a reference to the type `BOOLEAN` is a reference to the predefined type denoted by the pervasive identifier `BOOLEAN`, and not to any local redefinition that might have been made.

3.3.4 Interpretation of non-VDM-SL notations

The definition modules that are used in the specification of library modules shall be interpreted according to the concrete syntax, declaration semantics, static semantics, and dynamic semantics of definition modules as specified in section 6. Modula-2 comments that appear in these definition modules shall be interpreted as notes.

The pseudo-definition modules that are used in the specification of system modules shall be interpreted as notes which outline the declaration semantics of the facilities provided by the system modules and the static semantics that apply to their use. (The declaration semantics, static semantics, and dynamic semantics are defined explicitly in the relevant clauses of section 7.)

Natural language that appears between question marks on the right hand of syntax production rules shall be interpreted as normative requirements.

Where the terms ‘implementation-defined’ and ‘implementation-dependent’ occur (in whatever notation, with or without hyphens), the requirements of section 4 (particularly documentation requirements) shall apply. For example, given the VDM-SL expression `implementation-defined: Char-set` section 4 requires implementors to provide the set of characters and to document any elements in that set.

3.3.5 References to formal parameter names

In order to allow succinct descriptions of the meaning of procedures in standard library modules, a convention has been adopted by which explanation of requirements is given in terms of a procedure call with the names of formal parameters in the place of actual parameters. Hence the description refers to formal parameter names as if they denote variables; precision is provided in the VDM-SL definition which accompanies all natural language descriptions.

NOTE — For example, the module `Strings` provides a procedure having the following heading:

```
PROCEDURE Append (source: ARRAY OF CHAR; VAR destination: ARRAY OF CHAR);
```

The corresponding informal description of `Strings.Append` is:

The call

```
Append(source, destination)
```

shall create a new string value by appending the string value in `source` onto `destination`. If the length of the created string value exceeds the capacity of `destination`, the string value shall be truncated to the capacity of `destination`, and the truncated value shall be assigned to `destination`. If the length of the created string value is less than the capacity of `destination`, a string terminator shall be appended to the string value, and the resulting value shall be assigned to `destination`.

This simplifies the description of the value parameter `source` inasmuch as it can always be referred to by name, and no distinction need be made between literal values and non-constant expressions.

Section 4 : Requirements for Implementations

A conforming Modula-2 implementation meets the requirements for Modula-2 implementations that are laid down in this section.

A conforming standard library implementation meets the requirements for standard library implementations that are laid down in this section.

4.1 Translation

A conforming Modula-2 implementation shall accept compilation modules for translation from source code in the lexical form specified in section 5 (subject to the provisions of 4.2), in the syntactic form specified in section 6, and in the static semantic form specified in sections 6 and 7, as required for the type of implementation specified in 4.3, and subject to minimum implementation limits as specified in 4.11. A conforming implementation may accept for translation non-text representations that are deriveable from such source code.

A conforming Modula-2 implementation shall translate the compilation modules of a program in such a way that the execution of the program on the computer system is in accordance with the dynamic semantics in sections 6 and 7.

NOTES

- 1 In sections 5, 6 and 7 a formal definition of the syntax and semantics of Modula-2 programs is given. The definition does not explicitly state the requirements for Modula-2 implementations. The present section states the requirement for an implementation to implement the semantics in sections 6 and 7, while allowing for specified minimum limits on the size and complexity of programs to be translated, allowing for implementation restrictions on the processing of preferred lexical symbols, and requiring that implementation-defined values be defined.
- 2 Since VDM-SL is, by its mathematical nature, very precise, it has been necessary to choose one particular evaluation order in expressing the semantics, whereas at some points the natural language states that any evaluation order is permitted. The VDM-SL definition can thus be considered to be over-specification. When this situation arises, an annotation to the VDM-SL is provided to indicate such over-specification.

4.2 Source Code Representation

A conforming Modula-2 implementation shall:

- provide all the keywords and required symbols specified in section 5, and shall recognize identifiers and literals as specified in that section;
- provide the required left and right brackets;
- provide the preferred left and right brackets if its processor supports them;
- provide the required left and right braces;
- provide the preferred left and right braces if its processor supports them;
- provide the required case separator;
- provide the preferred case separator if its processor supports it;
- provide the required dereferencing operator;
- provide the preferred dereferencing operator if its processor supports it;
- process separators (white space, comments and source code directives) in such a way that they have no effect on the meaning of the program other than delimiting tokens, or, in the case of source code directives, by determining an implementation-defined value

- detect (and shall issue a warning concerning) a source code directive whose body is not recognized.

NOTES

- 1 This part of ISO/IEC 10514 does not specify how compilation modules be stored, nor how a conforming Modula-2 implementation accepts input — see 1.3.
- 2 The Lexis is formally expressed as a syntax that defines tokens in terms of characters. An implementation may accept a tokenized compilation module, that has been produced according to the syntax, rather than a stream of characters — see 4.1.

4.3 Ordering of Declarations

With respect to the two alternative predicates for declarations given in 6.2.7, a conforming Modula-2 implementation shall translate source code in one of two ways:

- a) If the implementation is of a type identified as declare-before-use-in-declarations (see also 4.9a), it shall accept all features of the language; i.e. the first predicate in 6.2.7 shall apply.
- b) If the implementation is of a type identified as declare-before-use (see also 4.9a), it shall reject any program not satisfying the second predicate in 6.2.7 and during translation shall give a unique indication of the reason for rejection.

NOTES

- 1 Clause 6.2.7 contains two alternative predicates for declarations that encapsulate the different approaches that implementors may choose to support. Hence, so-called 'multi-pass' implementations (i.e. those that require programs to observe a 'declare-before-use-in-declaration' rule) will support the predicate *wf-total-declarations* that returns true and so-called 'single-pass' implementations (i.e. those that require programs to observe a more strict 'declare-before-use' rule) will support the alternative predicate.
- 2 The requirement to provide a unique indication when the first predicate for declarations (in 6.2.7) is not satisfied does not preclude a conforming implementation from emitting other (possibly erroneous) indications first. The exact form of the required indication is not defined by this part of ISO/IEC 10514, due to the varying translation styles and user interfaces.
- 3 There is a requirement for all conforming implementations to accept and correctly process the **FORWARD** directive — see 6.2.11.1 and 6.2.11.2.

4.4 Predefined Entities

A conforming Modula-2 implementation shall provide all the predefined entities as specified in sections 6 and 7.

Implementations whose target computer system supports the addressing of 8-bit locations shall provide the type **BYTE** and the constant **LOCSPERBYTE** in the module **SYSTEM** as specified in 7.1. The names **BYTE** and **LOCSPERBYTE** shall not be used in any implementation of **SYSTEM** with meanings other than those specified in 7.1.

A conforming implementation may extend the system module **SYSTEM** by adding to the types, constants, variables or procedures that it exports.

NOTE — Whether or not an implementation does extend the system module **SYSTEM**, the other requirements of this section apply. In particular, since conforming implementations are required to process programs in the static semantic form specified in section 7, and execute programs in accordance with the dynamic semantics in section 7, no change may be made to the meaning of the types, constants, variables or procedures specified in 7.1.

4.5 Library Modules

A conforming Modula-2 implementation shall provide all the separate modules as specified in section 8 and those separate modules shall conform to the requirements for standard library module implementations.

If any or all of the separate modules provided with a conforming Modula-2 implementation have the same module names as a separate module specified in section 9, those separate modules shall conform to the requirements for standard library module implementations.

In a conforming standard library module implementation, the provided modules shall be the same as the modules of the same name in section 8 or section 9, and shall not be extended or reduced with respect to the types, constants, variables or procedures that they export.

NOTES

- 1 The requirement not to 'reduce' a module is meant to prevent features specified by this part of ISO/IEC 10514 from being omitted in an implementation.
- 2 A conforming Modula-2 implementation or standard library implementation may provide additional modules, with names that are different from the names used in section 8 and section 9; for example an implementation may provide an additional device module, for use with the input/output library, to access the specific facilities of the filesystem of an underlying operating system.
- 3 A conforming Modula-2 implementation or standard library implementation may not provide any module that has the same name as one of those modules specified in sections 8 and 9, but which is different from that module. For example, an alternative strings module may not be called **Strings**.
- 4 With the exception of the module **Semaphores**, none of the operations in the separate modules specified in sections 8 and 9 is required to be atomic in the presence of processes or coroutines. For example, suppose the input stream to a program were

Hello Professor, we hope you are well.

and two concurrent calls of **TextIO.ReadToken** (from the I/O library in 9.2.2.1.2) were made in this program for input to distinct character arrays. It is not required that, after the calls are complete, one array contain **Hello** and the other **Professor**. Such a program is incorrect (because it shares a common resource between concurrent activities without arranging for mutual exclusion between them); its behaviour is not defined.

- 5 In some cases (in the definition of the I/O library and elsewhere) in the interests of clarity of exposition, the formal definition makes lavish use of resources (e.g. by constructing a large set of items, only to pick one and discard the rest, or by using a buffer as large as the file that is attached to the channel). The implementation is not expected to mimic this specification exactly. Indeed, providing that the specified meaning is achieved, a more parsimonious approach is to be encouraged.

4.6 Errors

A conforming Modula-2 implementation shall determine whether or not a program contains errors and shall indicate the result of this determination to the user of the implementation. If the implementation has not attempted to discover all errors, this fact shall also be indicated to the user.

In standard mode a conforming implementation shall treat the use of extensions as errors.

NOTES

- 1 Whereas a conforming implementation may extend the system module **SYSTEM** (see 4.4), a program that uses such an extension, and hence relies on a definition of the extension is not given meaning by this part of ISO/IEC 10514 — see 1.3.
- 2 While it is expected that implementations will not issue misleading error indications, it is likely that declare-before-use implementations, when issuing error indications just prior to reporting rejection of a program not satisfying the second predicate in 6.2.7, may issue misleading indications before a declare-before-use error is properly detected and reported. (See also 4.3.)

4.7 Exceptions

A conforming Modula-2 implementation shall treat the occurrence of an exception in the following manner:

- a) In standard mode it shall detect (either before or during execution) the occurrence of all language exceptions whose detection is mandatory (specified by the use of the function *mandatory-detection* in sections 6, 7, 8, and 9).

- b) It shall be free to detect, or not to detect, language exceptions whose detection is not mandatory (specified by use of the function *non-mandatory-detection* in sections 6, 7, 8, and 9). The documentation requirements of 4.9f shall apply to any modes of use that may be utilized to provide such detection.

NOTE 1 — The language exceptions that must be detected by an implementation and the functions *mandatory-detection* and *non-mandatory-detection* are given in 7.5.4.

A conforming standard library module implementation shall treat the occurrence of an exception in the following manner:

- a) In standard mode it shall detect (either before or during execution) all exceptions defined by standard library modules whose detection is mandatory (as specified in sections 8, and 9).
- b) It shall be free to detect and report, or not to detect, exceptions defined by required separate modules and supplied standard library modules whose detection is not mandatory (as specified in sections 8, and 9). The documentation requirements of 4.9f shall apply to any modes of use that may be utilized to provide such detection.

During exceptional termination, a Modula-2 implementation, where appropriate, shall propagate an indication of the raised exception to the environment from which the program was invoked.

NOTES

- 2 It is expected that implementations will allow users to choose whether or not to detect exceptions by selecting some 'mode of use'. To conform to this part of ISO/IEC 10514 an implementation must provide a standard mode in which it will detect all exceptions whose detection is mandatory.
- 3 The requirement to propagate an indication to the environment from which the program was invoked when appropriate is designed to avoid reference to the program user (since there may not be one, for example, in an embedded system application). It is accepted that indicating an exception from an embedded system application may have no meaning. In the formal definition, this requirement is captured by the function *exception-actions* in 6.12.11. The indication may or may not be derived from the implementation-defined text string that is supplied when an exception is raised.
- 4 There is no implied requirement that implementors provide a 'trace-back' facility during exceptional termination.

4.8 Implementation-dependencies

A conforming Modula-2 implementation shall be free whether to detect and report, or otherwise process, any reliance upon a definition of implementation-dependent values or implementation-dependent behaviour (specified in sections 6 to 10).

NOTE — A program that relies upon a definition of an implementation-dependent value or implementation-dependent behaviour, such as the order of evaluation of parameters, is not given meaning by this part of ISO/IEC 10514 — see 1.3. However, few production translators will be able to detect this situation reliably, and so 4.8 cannot be more specific about the processing of such a program.

4.9 Documentation

A conforming implementation shall be accompanied by documentation that shall include the following:

- a) a statement as to whether the implementation supports the 'declare-before-use-in-declarations' or 'declare-before-use' semantics (see 4.3 and 6.2.7);
- b) a statement on how to use the implementation in standard mode;
- c) a definition of all implementation-defined values, including rules for the determination of values that are not constants;
- d) rules for ascertaining the amount of workspace that is needed for a single activation of a particular procedure (without taking into account the workspace needed by any procedure that it calls);

- e) a statement defining the layout strategy in mapping components of a variable of a structured (composite) type onto the addressable locations of the target computer system.
- f) a statement defining the exception-detection capability of the implementation; this statement shall specify the influence of any relevant implementation modes (or options) that are provided;
- g) a general statement about the processing of programs that rely upon a definition of an implementation-dependent value or of implementation-dependent behaviour, especially where such uses are treated as undetected errors or exceptions;
- h) a statement describing any features accepted by the implementation that are extensions, and a statement on how it is possible to detect the use of such extensions, or prevent their use;
- i) a statement describing any additional separate modules that are supplied (i.e. modules other than those described in sections 8, and 9);
- j) a statement listing the names of any additional separate modules (i.e. modules other than those described in sections 8, and 9) that are used by the implementation of the standard library modules that are provided.

NOTE 1 — Requirement 4.9j is made so that the user of an implementation is aware of potential clashes with the names of other separate modules used in a program.

- k) information about the facilities that are available to enable users to produce equivalent versions of all the standard library modules that are provided with the implementation, and the standard library modules that are not provided with the implementation but for which the information is relevant to the type of the implementation.

NOTE 2 — Requirement 4.9k is to be interpreted such that, for example, cross-translators for embedded systems applications do not need to be accompanied by information on facilities to implement, say, the standard input/output library, if that library is not itself provided with the implementation. The intention is to allow the separate modules to be replaceable, even if they are modules such as input/output modules that need to access an underlying system, without requiring the facilities for such access to be specified by this part of ISO/IEC 10514.

4.10 Statement of Compliance

A conforming implementation shall be accompanied by a compliance statement, conforming to the requirements below. In the following, the text <This implementation> shall be replaced by an unambiguous name identifying the implementation.

If the implementation complies in *all* respects with the requirements of this part of ISO/IEC 10514, the compliance statement shall be:

"<This implementation> complies with the requirements of ISO 10514-1."

If the implementation complies with some but not all of the requirements of this part of ISO/IEC 10514, the compliance statement shall be:

**"<This implementation> partially complies with the requirements of ISO 10514-1.
The details of the non-conformities are as follows:"**

followed by a complete list of the requirements of this part of ISO/IEC 10514 with which the implementation does not comply.

4.11 Minimum requirements

A conforming implementation shall allow a value of a type declared as SET OF CHAR to be a value that includes all values of the character type that are provided by the implementation.

If an implementation places a restriction on the number of values of the base type of a set type such that the number of values can be at most N , then the implementation shall allow the base type to be any ordinal type that has at most N values.

If an implementation places a restriction on the number of values of the base type of a packedset type, such that the number of values can be at most N , then the implementation may restrict the base type to be a type whose values have ordinal numbers in the range 0 to $(N - 1)$.

An implementation may impose restrictions on the types that are allowed as the first parameter in calls of `SYSTEM.ROTATE` and `SYSTEM.SHIFT`, provided that at least the type `BITSET` is accepted.

A conforming implementation shall be such that the limits it imposes on the attributes specified in normative annex A shall be at least as great as the minimum limits specified in that annex (which are summarized in the table below).

NOTES

- Annex A provides Modula-2 *limit-specifier programs* to generate programs which, when the limit-specifier programs have been parameterized by defining the constant `MinLimit`, a conforming implementation must successfully translate and execute. Thus the annex specifies each attribute using a Modula-2 program and specifies the minimum limit an implementation may impose on the attribute in the same program.
- The table below summarizes the attributes and minimum limits specified by the limit-specifier programs in annex A. Where a number is marked *** that number is the value of the minimum limit for an attribute only for implementations in which `SYSTEM.BITSPERLOC * SYSTEM.LOCSPERWORD` (the size in bits of `SYSTEM.WORD`) is greater than or equal to 16; for implementations in which `SYSTEM.BITSPERLOC * SYSTEM.LOCSPERWORD` is less than 16, the value of the minimum limit for such an attribute is implementation-defined.

Clause giving limit-specifier program	General description of attribute	Value of MinLimit
A.1.1	length of identifiers	32
A.1.2	length of string literals	32 ***
A.1.3	size of single dimension arrays	4096 ***
A.1.4	number of dimensions of multidimension arrays	16
A.1.5	depth of nested records	16
A.1.6	number of sequential local modules	16
A.1.7	depth of nested local modules	16
A.1.8	number of enumeration literals	256 ***
A.1.9	number of sequential procedure declarations	16
A.1.10	depth of nested procedure declarations	16
A.1.11	number of coroutines	16 ***
A.1.12	number of parameters of a procedure	16 ***
A.1.13	depth of nested IF-THEN-ELSEs in THEN-branch	16
A.1.14	depth of nested IF-THEN-ELSEs in both branches	4
A.1.15	depth of nested LOOP statements	16
A.1.16	depth of nested WHILE statements	16
A.1.17	depth of nested REPEAT statements	16
A.1.18	depth of nested FOR statements	16 ***
A.1.19	depth of nested WITH statements	16
A.1.20	number of case alternatives with else part	255 ***
A.1.21	depth of nested parentheses in expressions	16
A.1.22	depth of recursion	256 ***
A.1.23	number of imported modules	16

- The precise meaning of the attributes listed above is given by the limit-specifier programs in annex A; the values above are the minimum which conforming implementations are required to be able to accept. Implementors are encouraged to provide for acceptance of values that are greater than the minimal limits wherever possible.
- Users should expect implementations to exhibit other limitations and should expect interactions when combining requirements.
- Executing a limit-specifier program will generate Modula-2 source code which has to be accepted (that is, translated and run without any unexpected errors occurring) by a conforming implementation. Except for those cases where it is the intention to specify multi-module requirements, generator programs are deliberately formulated such that there will be one program module generated per attribute.
- In processors involving both host and target systems, the values in the table without the *** are relevant to the capabilities of the host computer system, while those marked by *** relate to the value of `SYSTEM.BITSPERLOC * SYSTEM.LOCSPERWORD` of the target.

Section 5 : The Lexis

The Modula-2 Lexis is the specification of the lexical elements of Modula-2 source code including white space, comments, source code directives, and the tokens that are used in the concrete syntax. All the symbols and pervasive identifiers of Modula-2 are defined in this section.

5.1 Source Code Structure

modula2 source code = { delimited token }, { separator } ;

delimited token = { separator }, token ;

token = word token | symbol | constant literal ;

NOTE — A token is the largest sequence of characters that satisfies the definitions below. Consequently, a word token must be separated from a following word token, numeric literal, string literal or character number literal. Furthermore, a numeric literal must be separated from a following numeric literal or character number literal.

5.2 Word tokens

word token = identifier | keyword ;

5.2.1 Identifiers

identifier = portable identifier | full identifier | pervasive identifier ;

5.2.2 Portable Identifiers

portable identifier = (simple letter | low line), { simple alphanumeric | low line } ;

low line = "_" ;

CHANGE — The lowline symbol ('_') is permitted (and may be repeated) anywhere in an identifier.

5.2.3 Full Identifiers

full identifier = (national letter | low line), { national alphanumeric | low line } ;

5.2.4 Pervasive Identifiers

pervasive identifier =

"ABS"	"BITSET"	"BOOLEAN"	"CARDINAL"
"CAP"	"CHR"	"CHAR"	"COMPLEX"
"CMPLX"	"DEC"	"DISPOSE"	"EXCL"
"FALSE"	"FLOAT"	"HALT"	"HIGH"
"IM"	"INC"	"INCL"	"INT"
"INTERRUPTIBLE"	"INTEGER"	"LENGTH"	"LFLOAT"
"LONGCOMPLEX"	"LONGREAL"	"MAX"	"MIN"
"NEW"	"NIL"	"ODD"	"ORD"
"PROC"	"PROTECTION"	"RE"	"REAL"
"SIZE"	"TRUE"	"TRUNC"	"UNINTERRUPTIBLE"
"VAL" ;			

CHANGE — The following pervasive identifiers have been added to the language:

COMPLEX, CMLX, IM, INT, INTERRUPTIBLE, LENGTH, LFLOAT, LONGCOMPLEX, LONGREAL, PROTECTION, RE and UNINTERRUPTIBLE.

NOTE — Pervasive identifiers are not reserved words; if redeclared in a program, they will no longer have their predefined meaning in the scope of the redeclared identifier; see 6.2 and see also 6.11.2.

5.2.5 Keywords

keyword =			
"AND"	"ARRAY"	"BEGIN"	"BY"
"CASE"	"CONST"	"DEFINITION"	"DIV"
"DO"	"ELSE"	"ELSIF"	"END"
"EXIT"	"EXCEPT"	"EXPORT"	"FINALLY"
"FOR"	"FORWARD"	"FROM"	"IF"
"IMPLEMENTATION"	"IMPORT"	"IN"	"LOOP"
"MOD"	"MODULE"	"NOT"	"OF"
"OR"	"PACKEDSET"	"POINTER"	"PROCEDURE"
"QUALIFIED"	"RECORD"	"REM"	"RETRY"
"REPEAT"	"RETURN"	"SET"	"THEN"
"TO"	"TYPE"	"UNTIL"	"VAR"
"WHILE"	"WITH" ;		

CHANGE — The keywords EXCEPT, FINALLY, FORWARD, PACKEDSET, REM and RETRY have been added to the language.

NOTE — The keywords AND, DIV, IN, MOD, NOT, OR and REM are operator keywords; the rest are punctuation keywords.

5.3 Symbols and Operators

symbol = required symbol | symbol with alternatives | operator ;

5.3.1 Required Symbols

required symbol =			
colon	comma	ellipsis	equals
period	semicolon	left parenthesis	right parenthesis ;

colon = ":" ;

comma = "," ;

ellipsis = "..." ;

equals = "=" ;

period = "." ;

semicolon = ";" ;

left parenthesis = "(" ;

right parenthesis = ")" ;

5.3.2 Symbols with alternatives

symbol with alternatives =
left bracket | right bracket | left brace | right brace | case separator ;

left bracket = preferred left bracket | required left bracket ;

preferred left bracket = "[" ;

required left bracket = "(!" ;

right bracket = preferred right bracket | required right bracket ;

preferred right bracket = "]" ;

required right bracket = "!) " ;

CHANGE 1 — All implementations are required to provide '(!' and '!)' as alternatives to '[' and ']'.

left brace = preferred left brace | required left brace ;

preferred left brace = "{" ;

required left brace = "(: " ;

right brace = preferred right brace | required right brace ;

preferred right brace = "}" ;

required right brace = ":)" ;

CHANGE 2 — All implementations are required to provide '(:' and ':)' as alternatives to '{' and '}'.

case separator = preferred case separator | required case separator ;

preferred case separator = "|" ;

required case separator = "!" ;

CHANGE 3 — All implementations are required to provide '!' as an alternative to '|'.

5.3.3 Operators

operator =		
assignment operator	plus operator	set union operator
string concatenate symbol	minus operator	set difference operator
logical disjunction operator	multiplication operator	division operator
div operator	mod operator	rem operator
set intersection operator	symmetric set difference operator	logical conjunction operator
sign	logical negation operator	equals operator
inequality operator	less than operator	greater than operator
less than or equal operator	greater than or equal operator	subset operator
superset operator	set membership operator	dereferencing operator ;

assignment operator = ":@" ;

plus operator = "+" ;

set union operator = "+" ;

string concatenate symbol = "+" ;

CHANGE 1 — The string concatenate symbol has been added.

minus operator = "-" ;

set difference operator = "-" ;

logical disjunction operator = "OR" ;

multiplication operator = "*" ;

division operator = "/" ;

div operator = "DIV" ;

mod operator = "MOD" ;

rem operator = "REM" ;

set intersection operator = "*" ;

symmetric set difference operator = "/" ;

logical conjunction operator = and keyword | and synonym ;

and keyword = "AND" ;

and synonym = "&" ;

sign = identity operator | arithmetic negation operator ;

identity operator = "+" ;

arithmetic negation operator = "-" ;

logical negation operator = not keyword | not synonym ;

not keyword = "NOT" ;

not synonym = "~" ;

equals operator = "=" ;

inequality operator = not equals operator | not equals synonym ;

not equals operator = "<>" ;

not equals synonym = "#" ;

less than operator = "<" ;

greater than operator = ">" ;

less than or equal operator = "<=" ;

greater than or equal operator = ">=" ;

subset operator = "<=" ;

superset operator = ">=" ;

set membership operator = "IN" ;

dereferencing operator = preferred dereferencing operator | required dereferencing operator ;

preferred dereferencing operator = "^" ;

required dereferencing operator = "@" ;

CHANGE 2 — All implementations are required to provide '0' as an alternative to '^'

5.4 Constant Literals

5.4.1 Whole Number Literals

whole number literal = decimal number | octal number | hexadecimal number ;

decimal number = digit, { digit } ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

octal number = octal digit, { octal digit }, "B" ;

octal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" ;

hexadecimal number = digit, { hex digit }, "H" ;

hex digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "A" | "B" | "C" | "D" | "E" | "F" ;

5.4.2 Real Literals

real literal = digit, { digit }, ".", { digit }, [scale factor] ;

scale factor = "E", [sign], digit, { digit } ;

5.4.3 String Literals

string literal = quoted string | character number literal ;

quoted string =
 single quote, { quotable character - single quote }, single quote |
 double quote, { quotable character - double quote }, double quote ;

single quote = "'" ;

double quote = '"' ;

quotable character = national graphic character | space | white space control ;

NOTES

- 1 A quoted string cannot contain both single and double quote characters (since one or the other will mark the end of the string).
- 2 A quoted string cannot cross a line boundary.

character number literal = octal digit, { octal digit }, "C" ;

5.5 Separators

separator = white space | comment | source code directive ;

5.5.1 White Space

white space = space | new line | white space control ;

space = " " ;

new line = ? possibly empty implementation-defined character sequence ? ;

white space control = ? implementation-defined character sequence that shall not include new line ? ;

5.5.2 Comments

comment = "(*", comment body, "*)" ;

comment body = { national character | separator } ;

NOTE — Since comments may be nested, the addition of comment delimiters around program text that has quoted strings containing comment delimiters may result in a different program from that which was intended (and probably in an incorrect one).

5.5.3 Source Code Directives

source code directive = "<*", directive body, ">" ;

directive body = comment body ;

CHANGE — A syntax for source code directives has been introduced.

5.6 Letters and National Characters

CHANGE — The concept of what characters are permissible as letters (for use in identifiers) and what characters are permissible within quoted strings, comments and source code directives has been extended, so as to facilitate a widening variety of character sets.

national character = national graphic character | format effector | space ;

national graphic character =	
simple alphanumeric	national alphanumeric
implementation-defined graphic	punctuation character ;

simple alphanumeric = simple letter | digit ;

simple letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
| "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
| "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
| "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

national alphanumeric = national letter | national numeric ;

national letter = ? implementation-defined alphabetic character ? ;

national numeric = ? implementation-defined numeric character ? ;

implementation-defined graphic = ? implementation-defined character ? ;

punctuation character = ? implementation-defined punctuation character ? - national alphanumeric ;

format effector = ? implementation-defined format effector ? ;

Section 6 : The Language

This section describes the structure and meaning of a Modula-2 program by specifying its components and stating the rules governing their combination. The major components are the compilation modules: program modules, definition modules, and implementation modules. The components of these modules include import and export lists, definitions and declarations of constants, types, variables, procedures and local modules, and statements and expressions which use these entities.

There are some constants, types, and procedures that are predefined. Those with pervasive identifiers are described in this section. Those that must be accessed using import lists (from system modules) are described in section 7.

The meaning of an identifier at the point of its use is specified by the environment that applies at that point — see 6.11.1. An environment associates an identifier with an entity and with the properties of that entity. An environment that applies at a point of use of an identifier is constructed from the environment that applies in a textually enclosing module or procedure, from import lists, and from defining occurrences in definitions and declarations. Environments that are constructed to express the well-formedness conditions refer only to properties that are relevant to those conditions, such as the types of variables and parameters. Environments that are constructed to express the meaning of a program refer to properties required for execution, including the storage for variables and the blocks of procedures. If an identifier is not visible in a constructed environment, an environment applies in which only the pervasive identifiers are visible.

The meaning of a program is described by the effect that its execution has on the external state (which may include the state of data files). The effect of the executable components of a program is described by the changes to an internal program state which includes the values of variables — see 6.12.1.

6.1 Programs, Program Modules, and Separate Modules

6.1.1 Programs and Compilation Modules

The major components of a Modula-2 program are the compilation modules, so-called because they can be treated separately, although not independently, by an implementation. There is, however, no requirement that the treatment of compilation modules involves actual compilation.

A program module is a compilation module that may use entities exported by separate modules, but does not itself export entities for use by other modules. Separate modules may in turn use entities from other separate modules. The entities that are available for use by clients of a separate module are defined in a compilation module called a definition module. The definition module must exist before the well-formedness of client modules can be checked. This also imposes a constraint on the compilation order — the order in which compilation modules may be elaborated. A corresponding compilation module called an implementation module is needed to complete each separate module before the program can be executed. The actions that are taken on program execution are given in initialization and finalization bodies within the program module and within the implementation modules.

A Modula-2 program consists of a program module together with the separate modules it uses (and the separate modules they in turn use).

Concrete Syntax

compilation module = program module | definition module | implementation module ;

NOTE 1 — There is no production for a Modula-2 program, or for a separate module, because of the separate treatment of compilation modules.

Abstract Syntax**types**

Compilation-module = *Program-module*
 | *Definition-module*
 | *Implementation-module*;

Program :: *pmod* : *Program-module*
 dmods : *Definition-modules*
 imods : *Implementation-modules*;

Definition-modules = *Definition-module-set*;

Implementation-modules = *Implementation-module-set*

Static Semantics

For a particular program there shall be a single program module. For each separate module directly or indirectly imported by the program module, there shall be a single definition module and a single consistent corresponding implementation module — see 6.1.5.1.

NOTE 2 — This part of ISO/IEC 10514 defines the meaning of a program given the implementation modules supplied for the corresponding definition modules. This does not preclude an implementation from allowing prior selection from alternative implementation modules for a given definition module.

The identifier of the program module shall not be the same as the identifier of any separate module that forms part of the program.

NOTE 3 — The identifier of a separate module is the module identifier of its definition module and its corresponding implementation module.

The identifiers of the program module, and the identifiers of the separate modules, shall not be the same as the identifier of one of the system modules.

The use of separate modules by compilation modules shall be such that there exists at least one complete and valid order for the separate treatment of the compilation modules of the program — see 6.1.5.2.

NOTE 4 — If there are many possible orders that satisfy the constraints given in 6.1.5.2, this part of ISO/IEC 10514 does not define which order is chosen, since the choice has no effect on the meaning of the program.

The outermost environment for checking well-formedness shall be constructed from the environments defined by the definition modules that form part of the program, together with the environment defined by the system modules.

NOTES

- 5 The outermost environment for checking well-formedness may be understood in terms of a procedure within which all declarations are made by export from local modules which correspond to the separate modules and the system modules. The definition modules correspond to qualified export lists in those local modules.
- 6 The environment defined by the system modules contains the identifiers associated with the constants, types, and procedures available by import from the system modules — see section 7.

wf-program : *Program* → $\mathbb{B}$

wf-program (*mk-Program* (*pmod*, *dmods*, *imods*)) $\triangleq$
 let ρ_{out} = *outer-defs-environment* (*dmods*) in
 wf-modules (*pmod*, *dmods*, *imods*) ρ_{out} $\wedge$
 wf-program-module (*pmod*) ρ_{out} $\wedge$
 $\forall dmod \in dmods \cdot$ *wf-definition-module* (*dmod*) ρ_{out} $\wedge$
 $\forall imod \in imods \cdot$ *wf-implementation-module* (*imod*) ρ_{out} $\wedge$
 compilation-orders (*pmod*, *dmods*, *imods*) $\neq \{ \}$

26
25
27
29
30

wf-modules : *Program-module* × *Definition-modules* × *Implementation-modules* → *Environment* → $\mathbb{B}$

wf-modules (*pmod*, *dmods*, *imods*) $\rho \triangleq$
 $pmod.name \notin SYSTEM-MODULE-NAMES \wedge$
 $(\forall dmod \in dmods \cdot dmod.name \neq pmod.name \wedge$
 $\exists! imod \in imods \cdot dmod.name = imod.name \wedge$
 $dmod.name \notin SYSTEM-MODULE-NAMES \wedge$
 $wf-module-pair(dmod, imod)\rho) \wedge$
 $(\forall imod \in imods \cdot \exists! dmod \in dmods \cdot imod.name = dmod.name) \wedge$
 $wf-abstract-modules(imods)$

34

25

wf-abstract-modules : *Implementation-modules* → $\mathbb{B}$

wf-abstract-modules (*imods*) $\triangleq$
 $\forall mk-Abstract-implementation-module(name_1, -, -, dens_1, -),$
 $mk-Abstract-implementation-module(name_2, -, -, dens_2, -) \in imods \cdot$
 $name_1 \neq name_2 \Rightarrow dom dens_1 \cap dom dens_2 = \{ \}$

annotations The correspondence of definition and implementation modules and the correctness of each of the compilation modules are checked with respect to the definitions of the definition modules and the system modules.

Dynamic Semantics

The program state shall be initialized from the external state as it exists before execution of the program. The initial protection from interrupts shall be implementation-defined.

NOTES

- 7 No variables are defined in this initial program state, and none exist upon completion of the program.
- 8 The external state includes the storage locations that a Modula-2 program can have access to via an address that is calculated from the machine address component of a variable declaration — see 6.12.8.2.1.

The outermost environment for program execution shall be constructed from the environments defined by the separate modules that form part of the program, together with the environment defined by the system modules.

NOTES

- 9 The outermost environment for program execution may be understood in terms of a procedure within which the program module, separate modules, and system modules are local modules. The definition modules correspond to qualified export lists in those local modules.
- 10 The environment defined by the system modules contains the identifiers associated with the constants, types, and procedures available by import from the system modules — see section 7.

The program shall be executed by first initializing the separate modules, in the order defined by their occurrence in the import lists, and then initializing the program module.— see 6.1.6, 6.1.4, and 6.1.2.

NOTE 11 — In a program with coroutines, this execution is performed by the main coroutine.

If an exception is raised during initialization, and this exception is not handled, the program shall be terminated exceptionally.

On program termination, the program module and the implementation modules shall be finalized in the reverse order of their initialization — see 6.1.7.

m-program : *Program* → *External-program-state* → *External-program-state*

m-program (*mk-Program* (*pmod*, *dmods*, *imods*))(*ext-state*) $\triangleq$
 $let initial-state = read-external-program-state(ext-state) in$
 $execute-program(pmod, dmods, imods) stop cenv initial-state$

331

26

annotations *stop*, *cenv*, and *initial-state* are supplied as explicit arguments in the application of *execute-program* as a function.
Corresponding arguments to these are implied in calls of operations.

execute-program : *Program-module* $\times$ *Definition-modules* $\times$ *Implementation-modules* $\xrightarrow{o}$ ()

```

execute-program (pmod, dmods, imods)  $\triangleq$ 
  def  $\rho_{out}$  = merge-modules(pmod, dmods, imods) in 26
  def  $\rho_{cont}$  = c-tize({EXCEPTION  $\mapsto$  (termination(); exception-actions())})  $\rho_{out}$  in 345 41 341
  (set-initializing(); 340
  let ex-order = execution-order (pmod, dmods, imods) in 39
  initialize-modules(ex-order)  $\rho_{cont}$ ; 26
  initialize-program-module(pmod)  $\rho_{cont}$ ; 28
  termination(); 41
  termination-actions() ) 341

```

Auxiliaries

outer-defs-environment : *Definition-modules* $\rightarrow$ *Environment*

```

outer-defs-environment (defs)  $\triangleq$ 
  let  $\rho$  = let  $\rho_{defs}$  = merge-environments ({d-definition-module (def)  $\rho$  | def  $\in$  defs}) in 319 29
    merge-environments ({ $\rho_{defs}$ ,  $\rho_{sys-modules}$ }) in 319
   $\rho$ 

```

annotations The definition modules of the program are taken to construct an environment corresponding to the definitions of the definition modules and the system modules.

merge-modules : *Program-module* $\times$ *Definition-modules* $\times$ *Implementation-modules* $\xrightarrow{o}$ *Environment*

```

merge-modules (pmod, dmods, imods)  $\triangleq$ 
  let order  $\in$  compilation-orders (pmod, dmods, imods) in 36
  merge-compilation-modules(order)  $\rho_{sys-modules}$  26

```

annotations The set of valid orders of compilation is produced and one of these orders is chosen arbitrarily. The operation is modelling the link-edit process in constructing the outer environment for execution and allocating storage.

merge-compilation-modules : *Module-order* $\rightarrow$ *Environment* $\xrightarrow{o}$ *Environment*

```

merge-compilation-modules (order)  $\rho$   $\triangleq$ 
  if order = []
  then return  $\rho$ 
  else def  $\rho_{new}$  = m-compilation-module(hd order)  $\rho$  in 26
    merge-compilation-modules(tl order)  $\rho_{new}$  26

```

annotations The declarations of each module are elaborated in order. The order is such that all the definitions of all modules used by a client module have been added to the environment before that client module is processed. Elaboration of the declarations allocates storage and builds an environment, and the resulting outer environment for the execution of the program is constructed.

m-compilation-module : *Compilation-module* $\rightarrow$ *Environment* $\xrightarrow{o}$ *Environment*

```

m-compilation-module (cm)  $\rho$   $\triangleq$ 
  (is-Program-module(cm)  $\rightarrow$  m-program-module(cm)  $\rho$ , 28
  is-Definition-module(cm)  $\rightarrow$  m-definition-module(cm)  $\rho$ , 30
  is-Implementation-module(cm)  $\rightarrow$  m-implementation-module(cm)  $\rho$ ) 31

```

initialize-modules : *Execution-order* $\rightarrow$ *Environment* $\xrightarrow{o}$ ()

```

initialize-modules (order)  $\rho$   $\triangleq$ 
  if order = []
  then skip
  else (initialize-an-implementation-module(hd order)  $\rho$ ; 31
    initialize-modules(tl order)  $\rho$ ) 26

```

annotations The implementation modules are initialized in the execution order defined by the imports.

6.1.2 Program Modules

A program module consists of a sequence of import lists followed by a module block, which comprises a sequence of declarations and an optional module body. Both of these sequences may be empty. The declarations may include the declaration of local modules.

The program module block is executed after the initialization of all implementation modules of the program.

A program module may have an interrupt protection specified in its heading. The statements and procedures of the module are executed under a protection from interrupts as specified by the protection expression — see 6.1.11.

Concrete Syntax

```
program module =
  "MODULE", module identifier, [ interrupt protection ], semicolon,
  import lists,
  module block, module identifier, period ;
```

```
module identifier = identifier ;
```

The two module identifier components of a program module shall be identical.

Abstract Syntax

types

```
Program-module :: name      : Identifier
                  imports   : Import-lists
                  block     : Module-block
                  protection : [Interrupt-protection]
```

Static Semantics

The program module shall not import itself. If present, the import lists of the program module shall be valid given the outermost environment — see 6.1.8. If present, the import lists and the declarations of the program module block shall be consistent with one another — see 6.1.8.5.

Given the identifiers of the system modules and the separate modules that are imported into the program module, and given the identifiers that are imported from those modules:

- the declarations of the module block of the program module shall be well-formed — see 6.2.6,
- the bodies of module and procedure declarations of the module block shall be well-formed — see 6.2.7,
- if present, the body of the module block shall be a well-formed module body — see 6.5.3.

If present in the program module heading, the interrupt protection shall be a valid interrupt protection in the context of the pervasive identifier environment only — see 6.1.11.

wf-program-module : *Program-module* → *Environment* → $\mathbb{B}$

wf-program-module (*mk-Program-module* (*name*, *imports*, *block*, *protection*)) $\rho_{out} \triangleq$
name $\notin$ *elems directly-imports* (*imports*) $\wedge$
wf-import-lists (*imports*) $\rho_{out} \wedge$ 42
let *mk-Module-block* (*decls*, -, -) = *block* in
consistent-imports (*imports*, *decls*) $\rho_{out} \wedge$ 46
let $\rho_{import} = d\text{-import-lists}$ (*imports*) ρ_{out} in 42
(*wf-declarations* (*decls*) $\rho_{import} \wedge$ 65
wf-declaration-bodies (*decls*) $\rho_{import} \wedge$ 67
wf-module-body (*block*) ρ_{import}) $\wedge$ 132
(*protection* = nil $\vee$ *wf-interrupt-protection* (*protection*) ρ_{empty}) 51

Dynamic Semantics

The elaboration of the program module shall elaborate the declarations of the module block, given the meaning of the imported identifiers.

If the program module is a protected module, the protection specified for the module shall be used to protect the protection domain of the program module. The protection shall be evaluated in the context of the pervasive identifier environment only.

m-program-module : *Program-module* → *Environment* $\xrightarrow{o}$ *Environment*

m-program-module (*mk-Program-module* (*name*, *imports*, *block*, *protection*)) $\rho \triangleq$
let $\rho_{import} = d\text{-import-lists}$ (*imports*) ρ in 42
def *pval* = if *protection* = nil then nil else *evaluate-interrupt-protection* (*protection*) ρ_{empty} in 51
let $\rho_{internal} = add\text{-protection}$ (*pval*) ρ_{import} in 318
def $\rho_{block} = m\text{-module-block}$ (*block*) $\rho_{internal}$ in 132
return *overwrite-mod-environment* ($\{name \mapsto \rho_{block}\}$) ρ 317

annotations The compilation order is such that, when this operation is called, the result of elaboration of all definition modules used by the definition module have been merged in the given environment. The protection expression (if present) is evaluated in an environment consisting of the pervasive identifiers only, and is added to the returned environment for use by the protection mechanism.

The program module shall be initialized by initializing the program module block given the environment resulting from the elaboration of the program module — see 6.5.4.

initialize-program-module : *Program-module* → *Environment* $\xrightarrow{o}$ *Environment*

initialize-program-module (*mk-Program-module* (*name*, -, *block*, -)) $\rho \triangleq$
let $\rho_{block} = associated\text{-module}$ (*name*) ρ in 317
execute-static-module-initialization (*block*) ρ_{block} 132

annotations The environment for initializing the module block of the program module is retrieved from the given environment.

6.1.3 Definition Modules

A definition module consists of a sequence of import lists and a sequence of definitions only. Both of these sequences may be empty. The identifiers introduced by the definitions denote entities that are to be available for use in other modules by import from the corresponding separate module — see 6.2.3.

In the case of procedure definitions, only the procedure headings are shown. This is sufficient to allow calls of these procedures in other modules to be checked for consistency, and for those modules to be prepared for execution.

In the case of type definitions, a complete declaration may be given, or the definition may specify only the type identifier, resulting in what is called an opaque type. If the type identifier is defined as the identifier of an enumeration type, the enumeration constant identifiers are also available for import from the corresponding separate module.

CLARIFICATION — This part of ISO/IEC 10514 adopts the position taken in later editions of *Programming in Modula-2* of not allowing export lists in definition modules.

Concrete Syntax

definition module =
 "DEFINITION", "MODULE", module identifier, semicolon,
 import lists, definitions,
 "END", module identifier, period ;

The two module identifier components of a definition module shall be identical.

Abstract Syntax

types

Definition-module :: *name* : *Identifier*
 imports : *Import-lists*
 defs : *Definitions*

Declaration Semantics

The export closure of the identifiers introduced by the definitions of a definition module, and the entities they denote, shall be associated with the identifier of the definition module in the outermost environment.

NOTE — The export closure includes the identifiers implicitly exported when an identifier is explicitly exported — see 6.1.10.

d-definition-module : *Definition-module* → *Environment* → *Environment*

d-definition-module (*mk-Definition-module* (*name*, *imports*, *defs*)) $\rho_{out} \triangleq$

let $\rho_{import} = d\text{-import-lists}$ (<i>imports</i>) ρ_{out} in	42
let $\rho_{def} = \text{overwrite-environment}$ (<i>d-definitions</i> (<i>defs</i>) ρ_{def}) ρ_{import} in	319 54
let <i>ids</i> = <i>identifiers-defined-in</i> (<i>defs</i>) in	82
let $\rho_{local} = \text{restrict-environment}$ (<i>ids</i>) ρ_{def} in	320
let <i>xids</i> = <i>x-closure</i> (<i>ids</i>) ρ_{def} in	49
let $\rho_{mod} = \text{restrict-environment}$ (<i>xids</i>) ρ_{def} in	320
<i>merge-environments</i> ({ <i>construct-module-environment</i> (<i>name</i> , ρ_{mod}),	319 317
<i>construct-local-environment</i> (<i>name</i> , ρ_{local})})	317

annotations The given environment is restricted by the import lists, and then used in the construction of the environment from the definitions of the definition module. The constructed module environment associates the name of the module with the export closure of the identifiers and entities defined in the module and is used for import from the module. The constructed local environment associates the name of the module with the identifiers and entities defined in the module, and is used within the corresponding implementation module.

Static Semantics

A definition module shall not import from its own separate module. If present, the import lists of a definition module shall be valid given the outermost environment — see 6.1.8. If present, the import lists and the definitions of a definition module shall be consistent with one another — see 6.1.8.4.

The definitions of the definition module shall be well-formed given the identifiers of the system modules and the separate modules that are imported into the definition module, and given the identifiers that are imported from those modules.

$wf\text{-}definition\text{-}module : Definition\text{-}module \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}definition\text{-}module (mk\text{-}Definition\text{-}module (name, imports, defs))\rho_{out} \triangleq$
 $name \notin elems\ directly\text{-}imports (imports) \wedge$
 $wf\text{-}import\text{-}lists (imports)\rho_{out} \wedge$ 42
 $consistent\text{-}imports\text{-}in\text{-}definitions (imports, defs)\rho_{out} \wedge$ 45
 $let\ \rho_{import} = d\text{-}import\text{-}lists (imports)\rho_{out}\ in$ 42
 $wf\text{-}definitions (defs)\rho_{import}$ 55

Dynamic Semantics

The elaboration of a definition module shall elaborate the definitions of the definition module, given the meaning of the imported identifiers.

$m\text{-}definition\text{-}module : Definition\text{-}module \rightarrow Environment \xrightarrow{o} Environment$
 $m\text{-}definition\text{-}module (mk\text{-}Definition\text{-}module (name, imports, defs))\rho \triangleq$
 $let\ \rho_{import} = d\text{-}import\text{-}lists (imports)\rho\ in$ 42
 $def\ \rho_{def} = m\text{-}definitions(defs)\ \rho_{import}\ in$ 56
 $let\ ids = identifiers\text{-}defined\text{-}in (defs)\ in$ 82
 $let\ \rho_{local} = restrict\text{-}environment (ids)\rho_{def}\ in$ 320
 $let\ xids = x\text{-}closure (ids)\rho_{def}\ in$ 49
 $let\ \rho_{mod} = restrict\text{-}environment (xids)\rho_{def}\ in$ 320
 $return\ merge\text{-}environments (\{\rho, \rho_{mod}, \rho_{local}\})$ 319

annotations The compilation order is such that, when this operation is called, the result of elaboration of all definition modules used by the definition module have been merged in the given environment.

6.1.4 Implementation Modules

The formal model in this part of ISO/IEC 10514 provides two ways by which the dynamic semantics of an implementation of a separate module may be determined:

- from the source text of a Modula-2 implementation module for the separate module,
- from a formal specification of the separate module, as given for the standard library modules in sections 8 and 9.

The meaning of a program that uses a standard library module is obtained by producing an abstract implementation module directly from the specification. The formal model of an implementation module for which Modula-2 source is submitted is called a sourced implementation module in this part of ISO/IEC 10514.

Abstract Syntax

types

$Implementation\text{-}module = Sourced\text{-}implementation\text{-}module \mid Abstract\text{-}implementation\text{-}module$

Static Semantics

$wf\text{-}implementation\text{-}module : Implementation\text{-}module \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}implementation\text{-}module (imod)\rho_{out} \triangleq$
 $(is\text{-}Sourced\text{-}implementation\text{-}module(imod) \rightarrow wf\text{-}sourced\text{-}implementation\text{-}module(imod)\rho_{out},$ 32
 $is\text{-}Abstract\text{-}implementation\text{-}module(imod) \rightarrow wf\text{-}abstract\text{-}implementation\text{-}module(imod)\rho_{out})$ 33

Dynamic Semantics

$m\text{-implementation-module} : \text{Implementation-module} \rightarrow \text{Environment} \xrightarrow{o} \text{Environment}$

$m\text{-implementation-module}(im)\rho \triangleq$

$(\text{is-Sourced-implementation-module}(imod) \rightarrow m\text{-sourced-implementation-module}(imod)\rho,$ 32

$\text{is-Abstract-implementation-module}(imod) \rightarrow m\text{-abstract-implementation-module}(imod)\rho)$ 33

$initialize\text{-an-implementation-module} : \text{Implementation-module} \rightarrow \text{Environment} \xrightarrow{o} ()$

$initialize\text{-an-implementation-module}(im)\rho \triangleq$

$(\text{is-Sourced-implementation-module}(im) \rightarrow initialize\text{-a-sourced-implementation-module}(im)\rho,$ 33

$\text{is-Abstract-implementation-module}(im) \rightarrow initialize\text{-an-abstract-implementation-module}(im)\rho)$ 33

6.1.4.1 Sourced Implementation Modules

An implementation module in source form consists of a sequence of import lists followed by a module block, which comprises a sequence of declarations and an optional module body. Both of these sequences may be empty. The source code for an implementation module is only distinguishable from the source code for a program module by the appearance of the keyword **IMPLEMENTATION** as the first token.

The meaning of procedures and opaque types defined in the definition module is expressed by full declarations in the implementation module. The declarations may include the declaration of local modules and other entities hidden from clients of the separate module.

After the initialization of the implementation modules of imported separate modules, any local modules declared in the implementation module are initialized and the initialization body of the implementation module (if any) is executed. This order of initialization allows actions to be taken before exported procedures are called from client modules and is guaranteed, provided that no separate module indirectly imports itself — see 6.1.6.

An implementation module may have an interrupt protection specified in its heading. The statements and procedures of the module are executed under a protection from interrupts as specified by the protection expression — see 6.1.11.

Concrete Syntax

implementation module =

"IMPLEMENTATION", "MODULE", module identifier, [interrupt protection], semicolon,
import lists,
module block, module identifier, period ;

The two module identifier components of an implementation module shall be identical.

Abstract Syntax

types

$Sourced\text{-implementation-module} :: name : Identifier$
 $imports : Import\text{-lists}$
 $block : Module\text{-block}$
 $protection : [Interrupt\text{-protection}]$

Static Semantics

An implementation module shall not import from its own separate module. If present, the import lists of an implementation module shall be valid given the outermost environment — see 6.1.8. If present, the import lists and the declarations of an implementation module block shall be consistent with one another — see 6.1.8.5.

NOTE — An implementation module and its corresponding definition module must also be consistent (for example the exported procedures must be declared) — see 6.1.1 and 6.1.5.1.

Given the identifiers of the system modules and the separate modules that are imported into an implementation module, given the identifiers that are imported from those modules, and given the identifiers introduced by the definitions of the corresponding definition module:

- the declarations of the module block of the implementation module shall be well-formed — see 6.2.6,
- the bodies of module and procedure declarations of the module block shall be well-formed — see 6.2.7,
- if present, the body of the module block shall be a well-formed module body — see 6.5.3.

If present in an implementation module heading, the interrupt protection shall be a valid interrupt protection in the context of the pervasive identifier environment only — see 6.1.11.

wf-sourced-implementation-module : *Sourced-implementation-module* $\rightarrow$ *Environment* $\rightarrow$ $\mathbb{B}$

wf-sourced-implementation-module (*mk-Sourced-implementation-module* (*name*, *imports*, *block*, *protection*)) ρ_{out} $\triangleq$

<i>name</i> $\notin$ <i>elems directly-imports</i> (<i>imports</i>) $\wedge$	42
<i>wf-import-lists</i> (<i>imports</i>) ρ_{out} $\wedge$	
let <i>mk-Module-block</i> (<i>decls</i> , -, -) = <i>block</i> in	
<i>consistent-imports</i> (<i>imports</i> , <i>decls</i>) ρ_{out} $\wedge$	46
let ρ_{def} = <i>associated-local-environment</i> (<i>name</i>) ρ_{out} in	317
let ρ_{import} = <i>d-import-lists</i> (<i>imports</i>) ρ_{out} in	42
let ρ_{block} = <i>merge-environments</i> ($\{\rho_{import}, \rho_{def}\}$) in	319
(<i>wf-declarations</i> (<i>decls</i>) ρ_{block} $\wedge$	65
<i>wf-declaration-bodies</i> (<i>decls</i>) ρ_{block} $\wedge$	67
<i>wf-module-body</i> (<i>block</i>) ρ_{block}) $\wedge$	132
(<i>protection</i> = nil $\vee$ <i>wf-interrupt-protection</i> (<i>protection</i>) ρ_{empty})	51

Dynamic Semantics

The elaboration of an implementation module shall elaborate the declarations of the module block, given the meaning of the imported identifiers and the meaning of the identifiers that are introduced by the definitions of the definition module.

If an implementation module is a protected module, the protection specified for the module shall be used to protect the protection domain of the implementation module. The protection shall be evaluated in the context of the pervasive identifier environment only.

m-sourced-implementation-module : *Sourced-implementation-module* $\rightarrow$ *Environment* $\xrightarrow{o}$ *Environment*

m-sourced-implementation-module (*mk-Sourced-implementation-module* (*name*, *imports*, *block*, *protection*)) ρ_{out} $\triangleq$

let ρ_{def} = <i>associated-local-environment</i> (<i>name</i>) ρ_{out} in	317
let ρ_{import} = <i>d-import-lists</i> (<i>imports</i>) ρ_{out} in	42
let ρ_{com} = <i>combine-definition-and-imported-environments</i> ($\rho_{def}, \rho_{import}$) in	320
let ρ_{mod} = if <i>protection</i> $\neq$ nil	
then <i>mark-exported-from-protected</i> (<i>visible-identifiers</i> (ρ_{def})) ρ_{com}	318 319
else <i>mark-exported-from-protected</i> ($\{\}$) ρ_{com} in	318
def <i>pval</i> = if <i>protection</i> = nil then nil else <i>evaluate-interrupt-protection</i> (<i>protection</i>) ρ_{empty} in	51
let $\rho_{mod'}$ = <i>add-protection</i> (<i>pval</i>) ρ_{mod} in	318
def ρ_{block} = <i>m-module-block</i> (<i>block</i>) $\rho_{mod'}$ in	132
return <i>overwrite-mod-environment</i> ($\{\text{name} \mapsto \rho_{block}\}$) ρ_{out}	317

annotations The compilation order is such that, when this operation is called, the result of elaboration of all definition modules used by the implementation module have been merged in the given environment. The protection expression (if present) is evaluated in an environment consisting of the pervasive identifiers only, and is added to the returned environment for use by the protection procedures.

The implementation module shall be initialized by initializing the implementation module block given the environment resulting from the elaboration of the implementation module — see 6.5.4.

initialize-a-sourced-implementation-module : *Sourced-implementation-module* $\rightarrow$ *Environment* $\xrightarrow{o}$ ()

initialize-a-sourced-implementation-module (*mk-Sourced-implementation-module* (*name*, -, *block*, -)) $\rho \triangleq$
 let $\rho_{block} = \text{associated-module}(\text{name})\rho$ in 317
 execute-static-module-initialization(*block*) ρ_{block} 132

annotations The environment for initializing the module block of the implementation module is retrieved from the given environment.

6.1.4.2 Abstract Implementation Modules

Abstract implementation modules are derived from the formal specification of a separate module and not from Modula-2 source text. The method of producing an abstract implementation module is given in normative annex B.

Abstract Syntax

types

Abstract-implementation-module :: *name* : *Identifier*
 init : [*Initialization-actions*]
 procs : *Procedures*
 dens : *Procdens*
 final : [*Termination-actions*]
 inv *mk-Abstract-implementation-module* (-, -, *procs*, *dens*, -) $\triangleq$
 rng *procs* $\subseteq$ dom *dens*

Static Semantics

wf-abstract-implementation-module : *Abstract-implementation-module* $\rightarrow$ *Environment* $\rightarrow \mathbb{B}$
wf-abstract-implementation-module (*aim*) $\rho \triangleq$
 true

Dynamic Semantics

m-abstract-implementation-module : *Abstract-implementation-module* $\rightarrow$ *Environment* $\xrightarrow{o}$ *Environment*
m-abstract-implementation-module (*mk-Abstract-implementation-module* (*name*, -, *pids*, *pdens*, -)) $\rho_{out} \triangleq$
 (*store-procedure-dens*(*dens*)) ; 340
 let $\rho_{def} = \text{associated-module}(\text{name})\rho_{out}$ in 317
 let $\rho_{abstract} = \mu(\rho_{empty}, \text{procs} \mapsto \text{pids},)$ in
 let $\rho_{block} = \text{merge-environment}(\rho_{def}, \rho_{abstract})$ in 319
 return *overwrite-mod-environment* ({*name* $\mapsto \rho_{block}$ }) ρ_{out}) 317

initialize-an-abstract-implementation-module : *Sourced-implementation-module* $\rightarrow$ *Environment* $\xrightarrow{o}$ ()

initialize-an-abstract-implementation-module (*mk-Abstract-implementation-module* (-, *init*, -, -, *final*)) $\rho \triangleq$
 (if *final* $\neq$ nil
 then *queue-termination*(*mk-Abstract-module-finalization* (*final*, ρ)) 341
 else skip;
 let *handler* = *current-exception-handler* (ρ) in 318
 init() *handler*) 33

6.1.5 Program Module and Separate Module Consistency

Consistency between the compilation modules of a program is achieved if there is consistency between the correspondingly named definition and implementation modules, and if there is at least one valid compilation order.

6.1.5.1 Definition Module and Implementation Module Consistency

The definition module and implementation module of a separate module are consistent if their definitions and declarations are consistent.

Static Semantics

The identifiers imported into the implementation module by the import lists of the implementation module shall be distinct from the the identifiers defined in the definitions of the corresponding definition module.

Except for opaque types and procedures, no identifier defined in the definition module of a separate module shall be declared in the corresponding implementation module.

Each opaque type defined in the definition module of a separate module shall be declared in its corresponding implementation module as a pointer type or as identical to another opaque type.

NOTES

- 1 The defining instance of the type corresponding to an opaque type identifier defined in a definition module need not be within the declarations of the corresponding implementation module — the opaque type identifier may be exported from a local module of the implementation module or the opaque type may be declared as identical to that denoted by an imported identifier.
- 2 An opaque type may be redeclared as the address type, since the address type is also a pointer type.

CHANGE — In the third edition of *Programming in Modula-2*, opaque types must be declared as pointer types. This part of ISO/IEC 10514 allows an opaque type to be declared as another opaque type, since that other opaque type must directly or indirectly be implemented as a pointer type.

CLARIFICATION 1 — An opaque type cannot be declared as a scalar type.

Each procedure heading in the definition module of a separate module shall have a matching procedure declaration in the corresponding implementation module.

NOTE 3 — A procedure declaration matching a procedure heading need not be within the declarations of the corresponding implementation module — the procedure identifier may be exported from a local module of the implementation module.

The formal parameter list of the procedure heading in the definition module shall match the formal parameter list of the procedure heading of the procedure declaration in the implementation module. The formal parameter lists of procedure headings shall match if the formal types of each of the value parameter specifications or variable parameter specifications of corresponding formal parameters are identical.

CLARIFICATION 2 — The definition of a procedure in a definition module and the corresponding declaration of that procedure in the implementation module must match in the sense that corresponding parameters must have the identical type. The identifiers used to denote the parameters need not be the same, and the identifiers used to denote the types of these parameters need not be the same.

NOTE 4 — Even if a definition module does not contain any procedure definitions or opaque type definitions there must still be a corresponding implementation module. Likewise, even if there are no exports from a module and the implementation module is used solely for initialization, there must still be a corresponding definition module.

$wf\text{-}module\text{-}pair : Definition\text{-}module \times Implementation\text{-}module \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}module\text{-}pair (dmod, imod)\rho \triangleq$

$(is\text{-}Sourced\text{-}implementation\text{-}module(imod) \rightarrow wf\text{-}sourced\text{-}module\text{-}pair (dmod, imod)\rho,$
 $is\text{-}Abstract\text{-}implementation\text{-}module(imod) \rightarrow wf\text{-}abstract\text{-}module\text{-}pair (dmod, imod)\rho)$

35