九州大学
KYUSHU UNIVERSITY

VDMTools

VDM-SL 言語マニュアル

How to contact:

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

VDM-SL 言語マニュアル *2.0*

— Revised for VDMTools v9.0.6

目 次

1 導入 **1**

2 準拠事項 **2**

3 具象構文表記法 **3**

4 データ型定義 **3**
4.1 基本データ型 4
4.1.1 ブール型 4
4.1.2 数値型 7
4.1.3 文字型 11
4.1.4 引用型 11
4.1.5 トークン型 12
4.2 合成型 13
4.2.1 集合型 13
4.2.2 列型 16
4.2.3 写像型 19
4.2.4 組型 24
4.2.5 レコード型 25
4.2.6 合併型と選択型 29
4.2.7 関数型 30
4.3 不変条件 33

5 アルゴリズム定義 **34**

6 関数定義 **35**
6.1 多相関数 39
6.2 高階関数 40

7 式 **41**
7.1 let 式 41
7.2 def 式 45
7.3 単項式または 2 項式 46
7.4 条件式 47
7.5 限量式 50
7.6 iota 式 52

7.7 集合式 . . . 53
7.8 列式 . . . 55
7.9 写像式 . . . 57
7.10 組構成子式 . . . 58
7.11 レコード式 . . . 59
7.12 適用式 . . . 60
7.13 ラムダ式 . . . 62
7.14 narrow 式 . . . 63
7.15 is 式 . . . 65
7.16 リテラルと名称 . . . 66
7.17 未定義式 . . . 68
7.18 事前条件式 . . . 69

8 パターン 70

9 束縛 76

10 値 (定数) 定義 77

11 状態定義 78

12 操作定義 81

13 文 87
13.1 let 文 . . . 87
13.2 def 文 . . . 90
13.3 ブロック文 . . . 91
13.4 代入文 . . . 92
13.5 条件文 . . . 95
13.6 for ループ文 . . . 97
13.7 while ループ文 . . . 100
13.8 非決定文 . . . 100
13.9 call 文 . . . 103
13.10 return 文 . . . 104
13.11 例外処理文 . . . 104
13.12 error 文 . . . 108
13.13 恒等文 . . . 109
13.14 仕様記述文 . . . 109

14 トップレベル仕様記述 **111**
14.1 フラット仕様記述 111
14.2 構造仕様記述 113
14.2.1 モジュールのレイアウト 113
14.2.2 輸出節 114
14.2.3 輸入節 119

15 動的リンクモジュール **121**

16 VDM-SL と ISO /VDM-SL の相違点 **125**

17 静的意味 **127**

A VDM-SL 構文 **130**
A.1 文書 130
A.2 モジュール 130
A.3 定義 133
A.3.1 型定義 134
A.3.2 状態定義 136
A.3.3 値定義 136
A.3.4 関数定義 137
A.3.5 操作定義 138
A.4 式 140
A.4.1 括弧式 141
A.4.2 ローカル束縛式 141
A.4.3 条件式 141
A.4.4 単項式 142
A.4.5 2 項式 144
A.4.6 限量式 147
A.4.7 iota 式 147
A.4.8 集合式 147
A.4.9 列式 147
A.4.10 写像式 148
A.4.11 組構成子式 148
A.4.12 レコード式 148
A.4.13 適用式 148
A.4.14 ラムダ式 149

A.4.15 narrow 式 149
A.4.16 is 式 149
A.4.17 未定義式 149
A.4.18 事前条件式 149
A.4.19 名称 150
A.5 状態指示子 150
A.6 文 150
A.6.1 ローカル束縛文 151
A.6.2 ブロック文と代入文 151
A.6.3 条件文 152
A.6.4 ループ文 153
A.6.5 非決定文 153
A.6.6 call 文と return 文 153
A.6.7 仕様記述文 153
A.6.8 例外処理文 154
A.6.9 error 文 154
A.6.10 恒等文 154
A.7 パターンと束縛 154
A.7.1 パターン 154
A.7.2 束縛 156

B 語彙 156
B.1 文字 156
B.2 記号 159

C 演算子優先順位 162
C.1 結合子のファミリー 163
C.2 適用子のファミリー 163
C.3 評価子のファミリー 164
C.4 関係子のファミリー 165
C.5 連結子のファミリー 166
C.6 構成子のファミリー 166
C.7 グループ化 167
C.8 型演算子 167

D 2 つの具象構文間の相違 167

E 標準ライブラリ **170**
E.1 数学ライブラリ 170
E.2 IO ライブラリ 172
E.3 VDMUtil ライブラリ 174

Index **176**

1 導入

この文書では、モジュール拡張を行なった、基本的には標準 ISO/VDM-SL [5] である、VDM-SL 言語の構文と意味定義を述べる[1]。また, 全ての構文上正しい VDM-SL 仕様記述は VDM-SL 内でも正しいことに注意する。私たちは、明確に、理解しやすい方法で言語を提供しようとしているが、この文書は完全な VDM-SL の参照マニュアルではない。言語の説明のより詳しいところは、文献[2]を参照している。VDM-SL 記法が VDM-SL 標準の表記法と異なっているところは全て、言うまでもなく、意味定義を入念に説明する。

VDM-SL 言語は VDM-SL Toolbox によってサポートされた言語である。([?]) この Toolbox は構文チェッカー、静的な意味定義チェッカー、インタープリタ[3] 、および C++へのコードジェネレータを含んでいる。一般的に、ISO/VDM-SL が非実行可能な言語であるので、インタープリタは言語の部分集合だけをサポートする。この文書は特に VDM-SL の意味定義がインタープリタで使用される意味定義と異なっている部分に焦点をあてていく。この文書内では、VDM-SL Toolbox からインタープリタについて言及するときは、いつも、「インタープリタ」という用語を使用していく。そして、ある言語の構成要素の意味定義が VDM-SL 標準と全く同じであるときは、”VDM-SL ”について言及していく

その結果、ASCII(また、置き換えと呼ばれる) 具象構文を使用するが、私たちは特別なキーワード字体ですべての予約語を表示していく。こうする理由は、その文書では ASCII 記法が入力として使用されている VDM-SL Toolbox の言語マニュアルとして機能するからである。数学的な具象構文は、Toolbox によって、自動的に生成することができ、より見やすい構文を作り出すことができる。

セクション 2 は、その言語がどうここで紹介されるのか、また、対応している VDM-SL Toolbox がどのように VDM-SL 標準に一致するのかを示す。

セクション 3 は構文の構成要素の記述に使用される BNF 記法を紹介する。VDM-SL の表記法はセクション 4 からセクション 14 で述べられる。セクション 16 は ISO/VDM-SL と VDM-SL の違いに関する全リストを提供する。さらにセクション 17 は VDM-SL の静的な意味定義の短い説明を含む。

[1] その他にもいくつかの拡張が含まれている。

[2] VDM-SL における証明が [4] と [1] で最もよく扱われるにもかかわらず、より多くの参考書は、[3] で与えられる

[3] Toolbox は清書機能、デバック機能、テスト適用範囲のサポートを提供するのに加わっているが、これらは基本的なコンポーネントである。

付録 A では、完全な言語の構文、付録 B では語彙詳細、付録 C では演算子の優先順位を紹介する。付録 D では数学の構文シンボルと ASCII 具象構文の違いをリストで紹介する。付録 E では、標準 ライブラリの詳細と使用方法を提供する。最終的には、文書内における全ての構文ルールの存在を定義するインデックスを提供する。

2 準拠事項

標準規格である VDM-SL には、いくつかの準拠レベルを記載する準拠事項の節がある。最も低レベルでの準拠事項は構文一致である。VDM-SL Toolbox では、標準 の構文記述に従った仕様記述を受け入れる。加えて、準拠条項に従えば排除されるべき多くの拡張記述 (第 16 章参照) についても受け入れる。

準拠事項レベル 1 では、恐らくの正しさに対する静的意味を扱う (第 17 章参照)。ここでは、恐らくは良形である標準記述を対象にし、他に多く存在するさまざまな記述は極力排除することとした[4]。

準拠事項レベル 2 およびこれ以降のレベル (最終レベル以外) では、絶対的な良形であることの静的意味チェックと、静的意味に加えられるいくつかのおこりうる拡張チェックについて扱う。絶対的に良形であることの検査機能は Toolbox が持っている。しかしながら、実際例においてはこれが最も価値あるものとは考えない。なぜなら「現実にある」例に対し、ほとんどすべての記述がこの検査をパスすることはないからである。

準拠最終レベルでは動的意味を扱う。ここで、標準の動的意味 (これは実行可能でない) からどのように逸脱しているかについては、添付書類を用いた詳細の提供が求められている。本書では、どのような構成要素が Toolbox にて翻訳されるか、ほんの少しの構成要素において逸脱するものが何であるかを説明し、この要求に事実上答える。このようにこの準拠レベルは、VDM-SL Toolbox の存在で条件が満たされている。

まとめれば、 VDM-SL はきわめて標準準拠に近いと言うことができるが、これを保証するに十分な時間は未だ費やされていない。

[4]例えば述語が存在する集合の理解において、標準記述では (恐らくは良形であることのチェックでは) 要素式に対する検査をまったく行わない。なぜなら、述語は false となる可能性がある (したがって全式が空集合を表すことになる可能性がある) からである。読者はこの例を実際試すことにも興味をもつであろうと確信する。

3 具象構文表記法

本書の中で、一部の言語構文については常にBNF表記を用いる。使用されるBNF表記法には、以下に示すような特殊記号が用いられる:

,	連結記号
=	定義記号
|	定義分離記号 (選択枝)
[ ]	オプションの構文項目を囲む
{ }	0回以上出現する構文項目を囲む
‘ ’	シングル引用リテラルは終端記号を囲むのに使用される
メタ識別子	非終端記号は小文字 (空白も含む) で記される
;	1つの規則の終わりを表わす終了記号
( )	グループ化に用いられる、つまり “a, (b | c)” は “a, b | a, c” と等しい。
–	終端記号の集合からの減算を表す (つまり “character – (‘"’)” はダブル引用リテラルを除くすべての文字を表す。)

4 データ型定義

伝統的なプログラミング言語と同様に、VDM-SL においてもデータ型を定義し適切な名称を与えることができる。例えば次のような等式が与えられたとする:

```
Amount = nat
```

ここでは “Amount(合計)” という名のデータ型を定義し、この型に属する値は自然数であると述べている (nat(自然数) は以下に記述される基本型の1つである)。VDM-SL の型体系で全般に共通する1つは、今この点について述べることは重要だが、相等と不等とはどのような値間にも用いることができるということである。プログラミング言語においてはしばしば、演算対象が同じ型であることを要求される。VDM-SL では合併型 (以下に示す) と呼ばれる構造があるので、これには当てはまらない。

この節では、データ型定義の構文について述べる。加えて、ある型に属する値はどのように構成され操作されうるのか (組込み演算子を用いて) について述べる。最初に基本データ型を示し、次に合成型へと進めよう。

4.1 基本データ型

以下にいくつかの基本型を提示する。その各々は次を含む:

- 構成の名称
- 構成の記号
- そのデータに属する特殊な値
- そのデータ型に属する値のための組込み演算子。
- 組込み演算子の意味定義。
- 組込み演算子の使用例[5]

組込み演算子の各々については、その意味定義の記述と共に、名称、記号、そして演算子の型が与えられる (ただし相等と不等の意味については、通常の意味に従うので、記述されていない。) 意味定義の記述において、識別子は例えば a, b, x, y 他 といったもので、対応する演算子型の定義で使用されるものを参照している。

基本型とは、言語により定義されていて、それ以上単純な値には分解することができない異なる値をもっている型とされる。主要な基本型として 5 つ：ブール型、数値型、文字型、トークン型、引用型 が挙げられる。以下にこの基本型について 1 つずつ説明していこう。

4.1.1 ブール型

一般的に VDM-SL では、その中で計算が終了しなかったり結果を出せなかったりするかもしれないシステムを対象とすることも許されている。このような潜在

[5] これらの例題中では、メタ記号 '≡' を用いて与えられた例題が何と同等であるかを示す。

的な未定義状態を取り扱うために、VDM-SL では 3 値論理：値は「true(真)」、「false(偽)」、「bottom/undefined(未定義)」のいずれかであるとする、を取り入る。インタプリターの意味定義は、演算対象の順番に重きをおかない LPF (Logic of Partial Functions、部分関数の論理) の 3 値論理 ([4] 参照) をもつものではないという意味において、 VDM-SL のものとは異なる。それでも、論理積 and、論理和 or、それに含意演算子は、最初の演算対象のみで結果を決定するのに十分であるならば、次の演算対象をあえて評価しようとはしない、という条件つきの意味定義をもつ。ある意味で、インタプリターの論理の意味定義は 3 値であると、VDM-SL に関してはまだ考えることができるであろう。しかしながら、未定義値は無限大ループやランタイムエラーになる可能性がある。

名称: ブール

記号: bool

値: true, false

演算子: 下記の a と b は任意のブール式を表す:

演算子	名称	型
not b	否定	bool → bool
a and b	論理積	bool ∗ bool → bool
a or b	論理和	bool ∗ bool → bool
a => b	含意	bool ∗ bool → bool
a <=> b	同値	bool ∗ bool → bool
a = b	相等	bool ∗ bool → bool
a <> b	不等	bool ∗ bool → bool

演算子の意味定義: 意味定義では、ブール値を扱う場合の <=> と = は等しい。and、 or、および =>においては条件つきの意味定義がある。⊥ によって定義されていない項目 (たとえば定義域外のキーをもつ写像に適用される) を表示しよう。ブール演算子に対する真理値表は次のとおり[6]:

[6]標準 VDM-SL ではこれらの真理値表は (=>以外は) 対称性をもつことに注目しよう。

否定 not b

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

論理積 a and b

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

論理和 a or b

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

含意 a => b

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

同値 a <=> b

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

例題: a = true で b = false と仮定すると次のとおり:

not a	≡	false
a and b	≡	false
b and ⊥	≡	false
a or b	≡	true
a or ⊥	≡	true
a => b	≡	false
b => b	≡	true
b => ⊥	≡	true
a <=> b	≡	false
a = b	≡	false
a <> b	≡	true
⊥ or not ⊥	≡	⊥
(b and ⊥) or (⊥ and false)	≡	⊥

4.1.2 数値型

数値型には5つの基本型：正の自然数、自然数、整数、有理数、そして実数がある。3つを除きどの数値演算子も、演算対象として5つの型の混在を許す。例外である3つとは、整数除算、法算、剰余算、である。

5つの数値型は階層構造をなし、実数 (real) が最も一般的な型で有理数 (rat)[7]、整数 (int)、自然数 (nat)、正の自然数 (nat1) と続く。

型	値
nat1	1, 2, 3, ...
nat	0, 1, 2, ...
int	..., -2, -1, 0, 1, ...
real	..., -12.78356, ..., 0, ..., 3, ..., 1726.34, ...

この表より、int ならばどのような数でも自動的に real であるが、nat であるとは限らないということがわかる。言い換えると、正の自然数は自然数の一部であり、その自然数は整数の、その整数は有理数の、有理数は最終的には実数の一部である、と表現することができる。次の表でいくつかの数が属する型を示す：

[7] VDM-SL Toolbox の見地からすれば 実数 (real) と 有理数 (rat) は違いがない。コンピューター上では有理数しか表現できないからである。

数	型
3	real, rat, int, nat, nat1
3.0	real, rat, int, nat, nat1
0	real, rat, int, nat
-1	real, rat, int
3.1415	real, rat

すべての数が必然的に real 型 (そして rat 型) であることに注意。

名称: 実数, 有理数, 整数, 自然数、そして 正の自然数

記号: real, rat, int, nat, nat1

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

演算子: 以下における x と y は数式を表すとする。これらの型について仮定はなされない。

演算子	名称	型
-x	負符号	real → real
abs x	絶対値	real → real
floor x	底値	real → int
x + y	加算	real * real → real
x - y	減算	real * real → real
x * y	乗算	real * real → real
x / y	除算	real * real → real
x div y	整数除算	int * int → int
x rem y	剰余算	int * int → int
x mod y	法算	int * int → int
x**y	べき算	real * real → real
x < y	より小さい	real * real → bool
x > y	より大きい	real * real → bool
x <= y	より小さいか等しい	real * real → bool
x >= y	より大きいか等しい	real * real → bool
x = y	相等	real * real → bool
x <> y	不等	real * real → bool

演算対象として書かれた型は、許される限りでの最も広範な型である。例えば負符号は5つのすべての型 (nat1, nat, int rat そして real) を演算対象と

する、ことを示している。

演算子の意味: 演算子であるマイナス符号、総和、差、積、商、小さい、大きい、等しいか小さい、等しいか大きい、相等関係、不等関係はこのような演算の通常の意味をもつ。

演算子名称	意味記述
底値	x と等しいかより小さい整数のうちで最大のもの
絶対値	x の絶対値、つまり x >= 0 ならば x そのままで x < 0 ならば -x となる
冪	x を y 回乗じたもの

整数商、剰余、そして法　が負の数にどのように作用するかについては、しばしば混乱がおきる。事実 -14 div 3 に対して有効な答えが２つある: Toolbox においてと同様-4 (the intuitive) となるか、たとえば Standard ML [6] においてと同様に-5 となるかである。したがってこれらの演算については詳細に説明しておくべきであろう。

整数除算は floor と実数除算を用いて定義される:

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

右辺の floor と abs の順により違いが生じ、その順を交換することで上記の例題は-5 となる。これは floor は常により小さい（か等しい）整数に従うからである、たとえば floor（14/3）は 4 である一方 floor（-14/3）は -5 である。

剰余 x rem y と　法 x mod y は、x と y の符号が同じであれば同じ値となるが、そうでない場合は異なる値となり、 rem は x の符号を mod は y の符号をとる。剰余と法の公式は次のとおり:

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

そのため, -14 rem 3 は -2 に等しく、 -14 mod 3 は 1 に等しい。実数軸をたどり、 -14 から進め 3 づつジャンプすることで、これらの結果を確認することができる。剰余はたどった負の数の最後の値であるが、それは x にあたる最初の引数が負であるからであり、一方の法はたどった正の数の最初の値であるが、それは y にあたる 2 番目の引数が正であるからである。

例題: a = 7, b = 3.5, c = 3.1415, d = -3, e = 2 とすると:

- a	≡	-7
abs a	≡	7
abs d	≡	3
floor a <= a	≡	true
a + d	≡	4
a * b	≡	24.5
a / b	≡	2
a div e	≡	3
a div d	≡	-2
a mod e	≡	1
a mod d	≡	-2
-a mod d	≡	-1
a rem e	≡	1
a rem d	≡	1
-a rem d	≡	-1
3**2 + 4**2 = 5**2	≡	true
b < c	≡	false
b > c	≡	true
a <= d	≡	false
b >= e	≡	true
a = e	≡	false
a = 7.0	≡	true
c <> d	≡	true
abs c < 0	≡	false
(a div e) * e	≡	6

4.1.3 文字型

文字型は、VDM 文字集合 (158 ページの表 11 を参照) 中の単一の文字すべてを含む。

名称: 文字

記号: char

値: 'a', 'b', ..., '1', '2', ...'+', '-' ...

演算子: 次の c1 と c2 は任意の文字を表す:

演算子	名称	型
c1 = c2	相等	char $*$ char $\to$ bool
c1 <> c2	不等	char $*$ char $\to$ bool

例題:

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

4.1.4 引用型

引用型は、パスカルのようなプログラミング言語においては列挙型に相当する。しかしながら VDM-SL においては、中括弧の中に様々な引用リテラルを書く代わりに引用型というシングル引用リテラルからなるものを用いて、それらを合併型の一部をなすものとする。

名称: 引用

記号: たとえば <QuoteLit>

値: <RED>, <CAR>, <QuoteLit>, ...

演算子: 以下の q と r が、列挙型 T に属する任意の引用値を表していると仮定すると:

演算子	名称	型
q = r	相等	$T * T \to$ bool
q <> r	不等	$T * T \to$ bool

例題: T を次に定義された型とする:

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

ここで a = <France>であるならば次のとおり:

```
<France> = <Denmark>              ≡ false
<SaudiArabia> <> <SouthAfrica>    ≡ true
a <> <France>                     ≡ false
```

4.1.5 トークン型

トークン型は、トークンと呼ばれる異なる値の可算無限集合からなる。トークンに対して実行される操作は、相等と不等のみである。VDM-SL, におけるトークンは、mk_token を用いて任意の式を囲む記述ができるのにもかかわらず、単独に表現することはできない。これが、トークン型を含む仕様のテストを可能にする方法である。しかしながら VDM-SL 標準に似せるためには、これらのトークン値はどんなパターンマッチングによっても分解できず、相等または不等の比較以外どのような演算にも用いることはできない。

名称: トークン

記号: token

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

演算子: 以下の s と t は任意のトークン値を表す:

演算子	名称	型
s = t	相等	token * token → bool
s <> t	不等	token * token → bool

例題: 次においてたとえば s = mk_token(6) 、 t = mk_token(1) とすると:

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

4.2 合成型

以下に合成型について記述する。各々は次を含む：

- 合成型定義の構文
- 構成要素をどのように用いるか示す等式
- この型に属する値をどのように構成するか示す例題ほとんどの場合に、基本構成子式の構文が与えられている前の節への参照が示される。
- この型に属する値に対する演算子 [8]
- 演算子の意味定義
- 演算子の使用例

演算子の各々に対し、名称、記号、演算子の型がその意味定義と共に与えられる (ただし相等と不等については、通常の意味に従うとして除かれる)。意味定義記述において、識別子はたとえば m, m1, s, s1 他 というような、対応する演算子型定義で用いられたものを参照する。

4.2.1 集合型

集合とは、値を順番をつけずに集めたものであり、それらはすべて同じ型のもので[9]、全体は 1 つとして扱われる。VDM-SL におけるすべての集合は有限である、なぜならばもともと有限個の要素しか含められないからだ。集合型の要素は任意の合成型でありうるし、例えば集合自身の集合であってもよい。

[8] これらの演算子は、第 7.3 節で全演算子が与えられるなかの単項式か 2 項式に用いられている。

[9] ただし合併型を用いれば、2 つの値に共通な型を見つけ出すのは常に可能であることに注意 (第 4.2.6 節参照)。

以下の記述には次の合意を用いる：A は任意の型、 S は集合型、 s、 s1、 s2 は集合値、 ss は集合値の集合、 e、 e1、 e2、 en は集合の要素、 bd1, bd2、 ...、 bdm は集合または型を示す識別子を束ねたもの、そして P は論理述語である。

構文:

```
型 = 集合型
   | ... ;

集合型 = 'set of', 型 ;
```

等式: S = set of A

構成子:

集合列挙: {e1, e2, ..., en} は列挙された要素の集合を構成する。空集合は {} と表記される。

集合内包: {e | bd1, bd2, ..., bdm & P} は、述語 P が true となるすべての束縛について式 e を評価することにより集合を定義する。束縛は集合束縛と型束縛のどちらかとなる[10]。集合束縛 bdn は pat1, ..., patp in set s という形式をもつが、ここでの pati はパターン (通常は単純な識別子である) であり、 s は 1 つの式で構成される集合である。型束縛も、in set がコロンに換わり s が型式となるという意味において、同様のものである。

すべての集合式に対する構文と意味定義は、第 7.7 節に与えられる

演算子:

[10]型束縛は実行可能ではないので一般的にはインタープリタで実行されない (これについては第 9 節を参照)。

演算子	名称	型
e in set s1	帰属	$A *$ set of $A \to$ bool
e not in set s1	非帰属	$A *$ set of $A \to$ bool
s1 union s2	合併	set of $A *$ set of $A \to$ set of A
s1 inter s2	共通部分	set of $A *$ set of $A \to$ set of A
s1 \ s2	差	set of $A *$ set of $A \to$ set of A
s1 subset s2	包含	set of $A *$ set of $A \to$ bool
s1 psubset s2	真包含	set of $A *$ set of $A \to$ bool
s1 = s2	相等	set of $A *$ set of $A \to$ bool
s1 <> s2	不等	set of $A *$ set of $A \to$ bool
card s1	濃度	set of $A \to$ nat
dunion ss	分配的合併	set of set of $A \to$ set of A
dinter ss	分配的共通部分	set of set of $A \to$ set of A
power s1	有限べき集合	set of $A \to$ set of set of A

A，set of A 型と set of set of A 型は単に型の構造を表すだけではないことに注意。たとえば、任意の集合 s1 と s2 の合併を行った場合、結果の集合の型は 2 つの集合型の合併型とすることができる。これについての例は第 4.2.6 節に与えられる。

演算子の意味:

演算子名称	意味記述
帰属関係	e が集合 s1 の要素であるかどうか検査する
非帰属関係	e が集合 s1 の要素でないことを検査する
合併	集合 s1 と s2 の合併、つまり s1 と s2 の両方の要素をすべて含む集合である。
共通部分	集合 s1 と s2 の共通部分、つまり s1 と s2 の両方にある要素を含む集合である。
差	s2 に含まれていない s1 の要素をすべて含む集合。s2 は s1 の部分集合である必要はない。
包含関係	s1 が s2 の部分集合であるかどうかを検査する、つまり s1 のすべての要素が s2 の要素であるかどうかである。どの集合もそれ自身の部分集合であることには注意。
真包含関係	s1 が s2 の真部分集合であることを検査する、つまり 部分集合でありしかも s2\s1 が空集合でないことである。

演算子名称	意味記述
濃度	s1 の要素の数。
分配的合併	結果の集合は ss のすべての要素 (それら自身が集合である) の合併である、つまり ss のすべての要素／集合のすべての要素を含む。
分配的共通部分	結果の集合はすべての要素の共通部分であり、つまり ss のすべての要素／集合の中の要素を含むということ。 ss は空集合であってはならない。
有限べき集合	s1 のべき集合である、つまり s1 のすべての部分集合の集合である。

例題: s1 = {<France>,<Denmark>,<SouthAfrica>,<SaudiArabia>}、s2 = {2, 4, 6, 8, 11} 、 s3 = {} であるときには以下のとおり:

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

4.2.2 列型

列値とは ある型の要素を順にならべた集まりで 1, 2, ..., n によって索引づけられるもの；ここでは n がこの列の長さとなる。列型とはある型の要素を有限個連続させた型であり、空列を含む場合 (空列を含む列型) と含まない場合 (空列を含まない列型) のいずれかとなる。列型の要素には任意の混在が許されている; た

とえばそれらが連続したものであればよいわけである。

以下はこの合意が用いられる: A は任意の型であり、L は列型であり、 S は集合型であり、 l, l1, l2 は列値であり、 ll は列値の列である。e1, e2 および en はこれらの列の要素、 i は自然数、 P は述語、 e は任意の式である。

構文:

型 = 列型
| ... ;

列型 = 空列を含む列型
| 空列を含まない列型 ;

空列を含む列型 = 'seq of', 型 ;

空列を含まない列型 = 'seq1 of', 型 ;

等式: `L = seq of A` または `L = seq1 of A`

構成子:

列列挙: `[e1, e2,..., en]` は、列挙された要素によって列を構成する。空列は `[]` と表現する。テキストリテラルは文字の列挙の簡約記法である (たとえば `"csk" = ['c','s','k']`)

列内包: `[e | id in set S & P]` は、述語 P が true となるようなすべての束縛に対して式 e を評価することで列を構成する。式 e は識別子 id を用いる。S は数の集合であり、 id は通常の順で数とマッチする (最小の数を最初として)

すべての列式の構文と意味定義については、第 7.8 節で述べる。

演算子:

演算子	名称	型
hd l	先頭	seq1 of $A \to A$
tl l	尾部	seq1 of $A \to$ seq of A
len l	長さ	seq of $A \to$ nat
elems l	要素集合	seq of $A \to$ set of A
inds l	索引集合	seq of $A \to$ set of nat1
l1 ^ l2	連結	(seq of A) $*$ (seq of A) $\to$ seq of A
conc ll	分配的連結	seq of seq of $A \to$ seq of A
l ++ m	列修正	seq of $A *$ map nat1 to $A \to$ seq of A
l(i)	列適用	seq of $A *$ nat1 $\to A$
l1 = l2	相等	(seq of A) $*$ (seq of A) $\to$ bool
l1 <> l2	不等	(seq of A) $*$ (seq of A) $\to$ bool

型 A は任意の型であって、連結や分配的連結の演算子に対する演算対象は、同じ型 (A) である必要はない。結果列の型は、複数の演算対象の型の合併型となる。第 4.2.6 節に例題が与えられている。

演算子の意味定義:

演算子名称	意味記述
先頭	l の最初の要素。 l は空列であってはならない。
尾部	l から最初の要素を取り除いた部分列。 l は空列であってはならない。
長さ	l の長さ。
要素集合	l の要素すべてを含む集合。
索引集合	l の索引すべてを含む集合。 {1,...,len l}。
連結	l1 と l2 の連結、つまり順に、 l1 の列要素のあとに l2 の列要素を続けた列。
分配的連結	ll の列要素 (これら自体が列である) が連結された列: 最初と第 2 の列要素を連結し、次に第 3 の列要素を連結し、等々。
列修正	列索引が m の定義域にある l の列要素は、その索引が写像された先の値域値に修正される。 dom m は索引 l の部分集合でなければならない。
列適用	l から始まる索引の要素。 i は l の索引集合になければならない。

例題: l1 = [3,1,4,1,5,9,2], l2 = [2,7,1,8],
l3 = [<England>, <Rumania>, <Colombia>, <Tunisia>] とすると以下のとおり：

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

4.2.3 写像型

A 型から B 型への写像型とは、 A (または A の部分集合) の要素各々を B の 1 つの要素と結合する型のことである。写像の値とは、この 2 つの要素の組を順不同で集めたものと考えることができる。各々の組の最初の要素をキーと呼ぶが、これは各組で最初の要素を用いて 2 番目の要素 (情報部分と呼ばれる) を得ることができるからである。よって 1 つの写像におけるキー要素は、すべて異なるものでなければならない。すべてのキー要素の集合をこの写像の定義域と呼び、一方すべての情報値の集合を値域と呼ぶ。VDM-SL におけるすべての写像とは有限のものである。写像型の定義域と値域の要素には任意の合成が許されていて、たとえば要素を写像とすることもできる。

特別な写像としては 1 対 1 写像がある。1 対 1 写像とは、値域の要素で 2 つ以上の定義域の要素と結合するものはない写像のことである。この 1 対 1 写像では、写像を逆にすることが可能である。

以下では次のとおりに用いる: m, m1、および m2 は、任意の A 型からもう 1 つの任意の B 型への写像を表し、ms は写像値の集合であり。a, a1, a2、および an は A

から取り出した要素である一方、 b, b1, b2 および bn は B から取り出した要素である。P は論理述語である。e1、e2 は任意の式であり、 s は任意の集合である。

構文:

型 = 写像型
| ... ;

写像型 = 一般写像型
| 1 対 1 写像型 ;

一般写像型 = 'map', 型, 'to', 型 ;

1 対 1 写像型 = 'inmap', 型, 'to', 型 ;

等式: M = map A to B または M = inmap A to B

構成子:

写像列挙: {a1 |-> b1, a2 |-> b2, ..., an |-> bn} は、列挙された写からなる写像を構成する。空写像は {|->} と表す。

写像内包: {ed |-> er | bd1, ..., bdn & P} は、述語 P が true と判断するすべてのありうる束縛上で、式 ed と er を評価することによって写像を構成する。

すべての写像式の構文と意味定義については、第 7.9 節で述べる。

演算子:

演算子	名称	型
dom m	定義域	(map A to B) $\to$ set of A
rng m	値域	(map A to B) $\to$ set of B
m1 munion m2	併合	(map A to B) $*$ (map A to B) $\to$ map A to B
m1 ++ m2	上書	(map A to B) $*$ (map A to B) $\to$ map A to B
merge ms	分配的併合	set of (map A to B) $\to$ map A to B
s <: m	定義域限定	(set of A) $*$ (map A to B) $\to$ map A to B
s <-: m	定義域削減	(set of A) $*$ (map A to B) $\to$ map A to B
m :> s	値域限定	(map A to B) $*$ (set of B) $\to$ map A to B
m :-> s	値域削減	(map A to B) $*$ (set of B) $\to$ map A to B
m(d)	写像適用	(map A to B) $* A \to B$
m1 comp m2	写像合成	(map B to C) $*$ (map A to B) $\to$ map A to C
m ** n	写像反復	(map A to A) $*$ nat $\to$ map A to A
m1 = m2	相等	(map A to B) $*$ (map A to B) $\to$ bool
m1 <> m2	不等	(map A to B) $*$ (map A to B) $\to$ bool
inverse m	逆写像	inmap A to $B \to$ inmap B to A

演算子の意味定義: ２つの写像 m1 と m2 は、dom m1 と dom m2 に共通の要素が両写像により同じ値に写像されるならば、両立している。

演算子名称	意味記述
定義域	m の定義域 (キーの集合)。
値域	m の値域 (情報値の集合)。
併合	m1 と m2 が結合した写像で、結果の写像は m1 と同様に dom m1 の要素に、 また m2 と同様に dom m2 の要素に、写像を行う。２つの写像は両立していなければならない。
上書	m1 に m2 を上書または併合する、つまり m1 と m2 は必ずしも両立する必要はないということを除けば、併合と似ている。共通の要素はいずれも m2 によるものとして写像される (したがって m2 は m1 を上書する)。
分配的併合	ms に含まれるすべての写像を併合することにより構成される写像。 ms に含まれる写像は両立していなければならない。

演算子名称	意味記述
定義域限定	m の要素のうちでキーが s に含まれるもの、から構成される写像をつくりだす。s は dom m の部分集合である必要はない。
値域限定	m の要素のうちで情報値が s に含まれるもの、から構成される写像をつくりだす。 s は rng m の部分集合である必要はない。
写像の適用	キーが d である写像の情報値。 d は m の定義域に含まれていなければならない。
写像の合成	m2 の要素に m1 の要素を合成してつくった写像。結果は m2 と同じ定義域をもった 1 つの写像である。あるキーに対応する情報値は、最初に m2 をキーに適用しその後 m1 をその結果に適用することによって見つけられるものである。rng m2 は dom m1 の部分集合でなければならない。
写像の反復	m からそれ自体を n 回繰り返すことで構成された写像。 n = 0 は dom m の各々の要素がそれ自体への写像である同一写像； n = 1 は m 自体である。 n > 1 に対して、 m の値域は dom m の集合でなければならない。
逆写像	m の逆写像。 m は 1 対 1 写像でなければならない。

例題: 次を仮定すると

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

以下のとおり：

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

rng m1   ≡ {1,2,4,9}
```

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

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

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


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

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

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

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

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

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

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

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

4.2.4 組型

組型の値を組と呼ぶ。組とは固定長のリストであり、組の i 番目の要素は組型の i 番目の要素に属さなければならない。

構文:

型 = 組型
| ... ;

組型 = 型, '*', 型, { '*', 型 } ;

組型は少なくとも 2 つの部分型から構成される。

等式： `T = A1 * A2 * ... * An`

構成子： 組構成子： `mk_(a1, a2, ..., an)`

組構成子についての構文と意味定義は第 7.10 節で述べられる。

演算子:

演算子	名称	型
`t.#n`	選択	$T * \mathsf{nat} \to Ti$
`t1 = t2`	相等	$T * T \to \mathsf{bool}$
`t1 <> t2`	不等	$T * T \to \mathsf{bool}$

組に対して有効な演算子は、構成要素選択、相等、不等、のみである。組構成要素は、選択演算子を用いたり組パターンとマッチングさせることで、アクセスすることもできる。組選択演算子についての意味定義の詳細および使用例は、第 7.12 節に述べる。

例題: `a = mk_(1, 4, 8), b = mk_(2, 4, 8)` とすると以下のとおり:

```
a = b          ≡  false
a <> b         ≡  true
a = mk_(2,4)   ≡  false
```

4.2.5 レコード型

レコード型は、プログラミング言語においての構造体に相当する。したがってこの型の要素は、前述の組型の節で述べられた組にいくぶんか似ている。レコード型と組型の違いは、レコードの異なる構成要素は相応の選択関数を用いることで、直接選択することができることである。さらに加えて、レコードは操作するとき用いられるべき識別子によってタグ付けされる。一般的な使い方として、タグを与えるためにはただ 1 つの項目からなるレコードも定義される。組とのもうひとつの違いとなるが、組は少なくとも 2 つの実体をもつ必要があるが、レコードは空でもよい。

VDM-SL, is_ は名称に対する予約接頭辞であって *is* 式の中で使用される。これは、あるレコード値がどのレコード型に属するのか決定するために用いられる、組込演算子である。しばしば合併型の部分型同士を区別することに用いられるため、更なる説明が第 4.2.6 節になされている。is_演算子は、レコード型を決定するのに加え、ある値が基本型のひとつであるかどうかの決定も行うことができる。

以下では次の約束に従う： `A` はレコード型、 `A1, ..., Am` は任意の型、 `r, r1, r2` はレコード値、 `i1, ..., im` はレコード値 `r` からの選択子、 `e1, ..., em` は任意の式である。

構文:

型 = レコード型
| ... ;

レコード型 = 'compose', 識別子, 'of', 項目リスト, 'end' ;

項目リスト = { 項目 } ;

項目 = [識別子, ':'], 型
| [識別子, ':-'], 型 ;

または省略型表記法で

レコード型 = 識別子, '::', 項目リスト ;

この識別子が表すものは型名かタグ名である。

等式:

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

または

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

または

```
A :: A1 A2
```

2 番目の表記では、 比較対象外 項目 が第 2 項 selsec に対し用いられる。この負符号は、等号演算子を使ってレコード比較を行うときにこの項が無視されることを指定している。最後の表記法では A の項目はひとつひとつに名称が付けられていないため、パターンマッチングによってのみ (組についてそうだったように) アクセスを行うことができる。

省略型表記である :: は前の 2 例でも使われ、タグ名が型名と等しいというものだが、この表記法は最もよく用いられている。より一般的である compose 表記法は、次のようにレコード型がそれより複雑な型の構成要素として直接記述されなければならない場合に、典型的に用いられる:

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

しかしながら、レコード型は型定義においてのみ用いることができるもので、例えば関数や操作に対するシグネチャにおいてではないことは明記しておくべきであろう。

レコード型は、合併型定義における代案 (4.2.6 を参照) として、典型的に用いられる:

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

ここで A と B は、自身がレコード型として定義されている。この状態で、is_ 前置詞を代案と区別するために用いることができる。

構成子: レコード構成子: mk_A(a, b) において、 a は型 A1 に属し b は型 A2 に属す。

すべてのレコード式に対する構文と意味定義は第 7.11 節で与えられる。

演算子:

演算子	名称	型
r.i	項目選択	$A * Id \to Ai$
r1 = r2	相等	$A * A \to$ bool
r1 <> r2	不等	$A * A \to$ bool
is_A(r1)	Ｉｓ	$Id * MasterA \to$ bool

演算子の意味定義:

演算子名称	意味記述
項目選択	レコード値 r の中で項目名が i である項目の値。r は i という名の項目をもっていなければならない。

例題: Score は以下のように定義される

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

さらに次の通りとする

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

このとき

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

':' の代わりに ':-' を用いて記述する比較対象外項目は、たとえばプログラム言語の抽象構文における低水準モデルにおいて役立つことがある。例としては、識別子の一意性に影響を与えることなく、それらの識別子の型に位置情報項目を加えたい場合などである。

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

この効果は pos 項が相等比較において無視されることにあり、たとえば次の例は true と評価されるであろう：

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

特にこのことは、以下の形の写像の典型的な環境において検索を行う場合に役に立つはずである:

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

このような写像は指定の識別子に対し最大 1 つの索引を含み、写像検索は pos 項目から独立したものとなる。

そのうえ、比較対象外項目は集合式に影響を与える。たとえば、

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

は次と等しくなる

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

ここにおける疑問符は 7 から 9 までを表している。

最後に比較対象外項目に対する有効なパターンとしては、don't care あるいは識別子パターンに限定されていることには注意しよう。比較対象外項目は 2 つの値を比較するときに無視されるものであり、それ以上複雑なパターンを用いることに対しては意味をなさないからである。

4.2.6 合併型と選択型

合併型は集合論理における和に相当する、つまり合併型として定義される型はその合併型の構成要素各々からすべての要素を含むことになる。合併型の中で互いに素であるとはいえない複数の型を用いることは、あまりよくない使用法だが可能ではある。しかし通常は、属する型として可能な複数の型から 1 つを考える場合には合併型が用いられる。合併型を構成する型としてしばしばレコード型がある。is_演算子を用いることで、合併型のある値がこういった型のいずれに属するものであるのかを決定することが可能である。

選択型 [T] とは合併型 T | nil に対してのいわゆる省略であり、この nil は値が存在しないことを表記するために用いられるものである。ただし集合 {nil} をひとつの型として用いることはできないので、nil を含む型のみが選択型となりうる。

構文:

型 = 合併型
| 選択型
| ... ;

合併型 = 型, '|', 型, { '|', 型 } ;

選択型 = '[', 型, ']' ;

等式: `B = A1 | A2 | ... | An`

構成子: なし

演算子:

演算子	名称	型
`t1 = t2`	相等	$A * A \to$ bool
`t1 <> t2`	不等	$A * A \to$ bool

例題: この例題中で Const, Var, Infix および Cond は省略 :: 記法を用いて定義されたレコード型であることから、Expr は合併型である。

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

また expr = mk_Cond(mk_Var("b",<Bool>),mk_Const(3), mk_Var("v",nil)) とすると:

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

合併型を用いることで、今までで定義してきた演算子の使用を拡張することができる。たとえば =を bool | nat 上でのテストと解釈することで次を得る。

```
1 = false  ≡  false
```

同様に、集合の合併や列の連結の代わりに合併型を用いることができる:

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

集合合併においては、 nat | bool 型の集合上での合併を考える；一方列連結に対しては、 char | <c> | <d>型の列を操作している。

4.2.7 関数型

VDM-SL では関数型もまた型定義に用いることができる。型 A (実際は型のリスト) から型 B への関数型というのは、型 A の各々の要素に対して B の要素を結びつける型である。関数の値は、プログラム言語においての関数と同じもので他に

副作用をおよぼすことのない (つまりグローバル変数を使用していない) ものとして考えることができる。

このような用い方は、関数が値として用いられるという意味で上級向けの使用法と考えることができる (したがって初読ではこの節はとばしていただいてもよい)。関数値は、ラムダ式 (以下を参照) によって生成されることもあるし、第 6 節に述べる関数定義による場合もある。関数値は、関数を引数としたりまた戻り値にすることができるという意味で、高階なものとなり得る。この方法を用いれば、最初のパラメーターの組が与えられると新しい関数が 1 つ返されるというように、関数はカリー化されることが可能である (次の例題を参照)。

構文:

型 = 関数型
| ... ;

関数型 = 部分関数型
| 全関数型 ;

部分関数型 = 任意の型, '->', 型 ;

全関数型 = 任意の型, '+>', 型 ;

任意の型 = 型 | '(',')' ;

等式: `F = A +> B`[11] または `F = A -> B`

構成子: 伝統的な関数定義に加えて、関数を構成する唯一の方法がラムダ式によるものである: `lambda pat1 : T1, ..., patn : Tn & body` ここにおける `patj` はパターン、`Tj` は型式、そして `body` は本体式で全パターンよりパターン識別子を用いることが許されている。

ラムダ式に対する構文や意味定義は、第 7.13 節にある。

演算子:

演算子	名称	型
`f(a1,...,an)`	関数適用	$A1 * \cdots * An \to B$
`f1 comp f2`	関数合成	$(B \to C) * (A \to B) \to (A \to C)$
`f ** n`	関数反復	$(A \to A) *$ nat $\to (A \to A)$
`t1 = t2`	相等	$A * A \to$ bool
`t1 <> t2`	不等	$A * A \to$ bool

[11] 全関数矢印は全定義関数のシグネチャにおいてのみ用いることができ、型定義においては用いることはできないことに注意したい。

型値間での相等と不等については、最大の注意を払うべきである。VDM-SL においてこれは、数学上の相等 (または不等) に相等するが、一般関数と同様に無限値に対して計算不能となる。このように、インタプリターでの相等は関数値の抽象構文上のものである (以下の `inc1` と `inc2` を参照)。

演算子の意味定義:

演算子名称	意味記述
関数適用	関数 `f` を a_j の値に適用した結果。第 7.12 章の適用式の定義を参照のこと。
関数合成	最初に `f2` を適用して次はその結果に `f1` を適用することと同等な関数。`f1` はカリー化されてもよいが，`f2` はいけない。
関数繰り返し	`f` を `n` 回適用することと同等な関数。`n = 0` の場合はそのパラメーター値をそのまま返す恒等関数となる。 `n = 1` の場合はその関数自身となる。 `n > 1` の場合、 `f` の戻り値はそれ自身のパラメーター型に含まれるものでなければならない。

例題: 以下に関数の値を定義してみよう:

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

ここで次のことが導かれる:

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

相等判定は、VDM-SL の意味定義に基づいての期待される結果に従うものではないことに注意したい。このように、関数といった無限値に対する相等の使用には**十分**注意深くなる必要がある。

4.3 不変条件

もし先に述べた等式によって指定されたデータ型が許されるべきでない値を含むような場合、それは 1 つの不変条件により 1 つの型の値に制限することができる。結果として、その型はもともとの値の部分集合に制限されるということである。このように、述語の手段によって、定義された型の条件にかなう値はこの式が true となる値に制限されるのである。

不変条件の使用についての一般的構成は次の通り:

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

ここで pat は Id 型に属する値にマッチングさせるパターンであり、expr は true となる式であり、パターン pat から識別子のいくつかまたはすべてを含んでいる。

ある不変条件が定義された場合、1 つの新しい (全) 関数がシグネチャと共に暗黙に生成される:

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

この関数は、他の不変条件、関数、あるいは操作の定義中で用いることも可能である。

たとえば、27 ページ上に定義されたレコード型 Score を思い返してみよう。不変条件を用いることで、得点数は勝つか引き分けたゲームの数と一致する、ということが保障できる:

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

この型に対して暗黙に作成される不変条件関数は次の通り:

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

5 アルゴリズム定義

VDM-SL では、アルゴリズムが関数と操作の両方により定義できる。しかしながら、伝統的なプログラム言語における関数にただちに相当するというものではない。VDM-SL において関数と操作を区別するものは、ローカルおよびグローバル変数の使用である。操作は、グローバル変数といくらかのローカル変数の両方を扱うことができる。ローカル変数とグローバル変数の両者については後に述べられる。関数は、グローバル変数にアクセスすることはできないしローカル変数を定義することも許されていないという意味で、純粋なものである。このように、操作が命令的なものである一方で、関数は純粋に作用的なものである。

関数と操作は、陽に (明確なアルゴリズム定義によって) あるいは陰に (事前条件または事後条件によって)、両方法で定義することができる。関数に対する明示的なアルゴリズム定義を式と呼ぶ一方、操作に対するそれは文と呼ぶ。事前条件は、関数や操作が評価される前に何を保持していなくてはならないかを指定する true の値をとる式である。事前条件は、パラメーター値と (操作の場合は) グローバル変数のみを参照することができる。事後条件もまた、関数や操作が評価された後に何が保持されなければならないかを指定する true の値をとる式である。事後条件は、結果識別子、パラメーター値、グローバル変数の現在値、そしてグローバル変数の旧値、を参照することができる。グローバル変数の旧値とは、操作が評価される前の変数の値のことである。関数ではグローバル変数の変更は許されていないが、操作だけはグローバル変数の旧値を参照することができる。

しかしながら、インタープリタにより関数と操作の両方の実行を可能にするためには、それらは明示的に定義されていなければならない[12]。VDM-SL では、陽関数および操作定義に対して追加の事前または事後条件を指定することもできる。陽関数および操作定義の事後条件において、結果の値は予約語 RESULT によって参照されなければならない。

[12]暗黙に指定された関数と演算は一般的に実行できない、というのもそれらの事後条件は出力を入力に明白に関係づける必要がないからである。出力が満たさなくてはならないプロパティを指定することで、しばしば済む。

6 関数定義

VDM-SL では、1 階関数と高階関数を定義することができる。高階関数とは、カリー化関数 (結果として関数を返す関数) かまたは関数を引数にとる関数である。さらには、1 階のものも高階のものもいずれも多相であることが可能である。一般的に、ある関数を定義するための構文は次の通り:

関数定義 = 'functions', [関数定義,
{ ';', 関数定義 }, [';']] ;

関数定義 = 陽関数定義
| 陰関数定義
| 拡張陽関数定義 ;

陽関数定義 = 識別子,
[型変数リスト], ':', 関数型,
識別子, パラメーターリスト, '==',
関数本体,
['pre', 式],
['post', 式],
['measure', 名称] ;

陰関数定義 = 識別子, [型変数リスト],
パラメーター型, 識別子型ペアリスト,
['pre', 式],
'post', 式 ;

拡張陽関数定義 = 識別子, [型変数リスト],
パラメーター型,
識別子型ペアリスト,
'==', 関数本体,
['pre', 式],
['post', 式] ;

型変数リスト = '[', 型変数識別子,
{ ',', 型変数識別子 }, ']' ;

識別子型ペアリスト = 識別子, ':', 型,
{ ',', 識別子, ':', 型 } ;

パラメーター型 = '(', [パターン型ペアリスト], ')' ;

パターン型ペアリスト = パターンリスト, ':', 型,
{ ',', パターンリスト,':', 型 } ;

関数型 = 部分関数型
| 全関数型 ;

部分関数型 = 任意の型, '->', 型 ;

全関数型 = 任意の型, '+>', 型 ;

任意の型 = 型 | '(',')' ;

パラメーター群 = '(', [パターンリスト], ')' ;

パターンリスト = パターン,{ ',', パターン } ;

関数本体 = 式
| 'is not yet specified' ;

ここで、あるモデルが発展過程にある間は is not yet specified の指定が関数本体として用いられることが許されている。

陽関数定義の簡単な例は関数 `map_inter` であり、これは自然数上の 2 つの両立する写像をもってきて、両者に共通する写を返すものである

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

関数結果についての主張を許すために選択事後条件をさらにまた用いることができることにも注意しよう:

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

同じ関数が暗黙的にまた定義されることも可能である:

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

拡張陽関数定義 (標準ではない) の簡単な例は関数 `map_disj` で、これは自然数上で 2 つの両立する写像を持ってきて、それらのどちらかの写像に対して唯一の写からなる写像を返す:

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

(ここにおいて事後条件をインタープリタに通す試みは、もしかすると実行時エラーを引き起こすかもしれない、というのは `m1(d)` と `m2(d)` は同時に両者が定義される必要はないからなのである。)

関数 map_inter と map_disj はインタプリターにより評価することが可能であるが、暗黙的関数である map_inter2 は評価することができない。しかしながら、これら 3 つの場合における事前条件と事後条件は他の関数のなかで使用することが可能である; たとえば map_inter2 の定義から、関数 pre_map_inter2 と post_map_inter2 を以下のシグネチャで得る:

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

これらの種類の関数は自動的にインタープリタで作成され、他の定義においても用いることができる (この技術は引用とよばれる)。一般的に、次のシグネチャをもつ関数 f に対して

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

関数に対する事前条件を定義することで、次のシグネチャの関数 pre_f が生成される。

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

そして関数に対する事後条件を定義することで、次のシグネチャの関数 post_f が生成される。

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

関数は再帰 (自分自身の呼び出し) を使って定義することもできる。再帰呼び出しを使う場合、'measure' 関数を追加することが推奨される。これによって、実行が終了すること保証する証明課題を生成することができるようになる。シンプルな階乗関数の例を以下に定義する:

```
functions

fac: nat +> nat
```

```
fac(n) ==
  if n = 0
  then 1
  else n * fac(n - 1)
measure id
```

ここで、 id は以下のように定義されている:

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

6.1 多相関数

関数はまた多相であることが可能である。これは、複数の異なる型の値のもとに使用可能な包括的な関数を生成することができるということを意味する。この目的のために、型引数 (または接頭辞@記号をおき通常の識別子と同様に記述された型変数) が用いられる。空のバッグをつくりだすための多相関数を考える: [13]

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

上記の関数が使用できる以前のこととして、関数 empty_bag の、たとえば整数といったある型のインスタンス生成を行わなくてはならない:

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

さぁこれで整数をいれるための新しいバッグをつくるために、関数 emptyInt を使用することができる。更なる多相関数の例としては:

[13] 多相関数の例は [2] から引用する。バッグというのは、バッグの中での要素からその要素の重複度への写像をモデル化したものである。ここでの重複度は少なくも 1 以上であり、つまり要素がないならこの写像の役目は負えないので、0 に写像されるものではない。

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

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

もし事前条件や事後条件が多相関数に対して定義された場合は、対応する述語関数もまた多相である。たとえばもし num_bag が下記のように定義されていたとすると

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

事前条件は次のようになるであろう

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

また、measure は機能が多相的に定義された時も使用されるべきである。ただし、現在は measure を高階関数では使うことができない。

6.2 高階関数

関数は他の関数を引数として受け取ることが許される。この簡単な例は、自然数の列となる関数 nat_filter であり、1 つの述語をもち、この述語を満足させる部分列を返すものである:

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

このとき`nat_filter (lambda x:nat & x mod 2 = 0, [1,2,3,4,5])` ≡ `[2,4]`. 実際、このアルゴリズムは自然数に限ったものではない、したがってこの関数の多相版を定義してもよいであろう:

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

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

関数はまた結果として関数を返してもよい。これの例は関数 `fmap` である:

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

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

7 式

この中の節では異なる種類の式の 1 つ 1 つについて述べていこう。各々を次の方法で記述する:

- BNF 構文記法
- 非公式な意味定義記述
- 使用の記述例

7.1 let 式

構文: 式 = let 式

```
    |   let be 式
    |   ... ;

let 式 = 'let', ローカル定義 { ',', ローカル定義 },
         'in', 式 ;

let be 式 = 'let', 束縛, [ 'be', 'st', 式 ], 'in',
            式 ;

ローカル定義 = 値定義
             |   関数定義 ;

値定義 = パターン, [ ':', 型 ], '=', 式 ;
```

ここでの構成要素である “関数定義” は第 6 節で述べられている。

意味定義: 単純な *let* 式 は次の形式をもつ:

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

ここで、p1，...，pn はパターン、e1，...，en はそれぞれの対応パターン pi にマッチさせる式であって、e は任意の型でよいが p1，...，pn の中のパターン識別子を含む式である。これは、パターン p1,...，pn が対応する式 e1，...，en とマッチさせられる文脈中での、式 e の値を示している。

ローカル関数定義を用いることで、より発展した形の let 式をつくることもできる。そのようなことを行う意味は単に、このようなローカル定義関数のスコープは let 式の本体に制限されているということにある。

標準の VDM-SL においては、定義の収集が相互に再帰するものとなる可能性がある。しかしながら VDM-SL においては、このようなものがインタープリタでサポートされることはない。さらに、すべての構成子が使用される前に定義されているように、定義に順番付けがされていなければならない。

let-be-such-that 式は次の形式をもつ:

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

ここでは、 b は集合値 (または型) に対する束縛で、 e1 は ブール式、 e2 は式だが何の型であってもよく、b におけるパターンのパターン識別子を含むものである。be st e1 部分はオプション。この式は、b のパターンが b の集合要素かまたは b の中の型の値とマッチさせる文脈中での式 e2 の値を示す[14]。 st e1 式がある場合は、マッチングの文脈中で e1 が true となる束縛のみが用いられる。

例題: *let* 式 は読みやすさの改善に役立つ、特に何回も使われる複雑な式は縮めることで改善される。たとえば 37 ページの関数 map_disj を改善することができる:

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

また複雑な構造体を構成要素に分解する上でも便利である。たとえば、前に定義したレコード型 Score (27 ページ) を使用することで、あるスコアがもうひとつより大きいかどうかをテストすることができる:

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

この特別な例では、2 つのスコアから 2 番目と 5 番目の構成要素を抽出している。don't care パターン (70 ページ) が、この式本体で行われた処理と残りの構成要素が無関係であることを示すために用いられていることに注目しよう。*let-be-such-that* 式 は、1 つの集合から 1 つの要素を選ぶ意味のない選択を減らすために、特に集合上での再帰定義の形式化において用いられる。これについての例は、列の filter 関数 (41 ページ) を集合上で考えたものである:

[14] 集合束縛のみはインタープリタによって実行できることを思い出そう。

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

別の方法として、この関数を集合内包 (第 7.7 節参照) を用いて定義することもできるであろう:

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

最後の例はオプションである “be such that” 部分をどのように用いることができるかを示す。いくつかのプロパティをもつある要素が存在することはわかっているがその要素に対する明示的な式がわからないまたは記述することが難しい場合に、この部分は特に役に立つ。たとえばこの式を選択ソートアルゴリズムを書くために活用することができる:

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

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

```
in [m]^(selection_sort (remove(m,l)))
```

ここでは、最初の関数は与えられたリストから与えられた要素を取り除く；2 番目の関数は並び替えされていないリスト部分から最も小さい要素を繰り返し取り除き、並び替えされた部分の頭に置く。

7.2 def 式

この式は、第 12 節で述べられる操作の内部でのみ用いることができる。式の部分でグローバル変数を取り扱うために、操作の内部で特別な式 (すなわち def 式) が許されている。

構文:

式 = ...
| def 式
| ... ;

def 式 = 'def', パターン束縛, '=', 式,
{ ';', パターン束縛, '=', 式 }, [';'],
'in', 式 ;

意味定義: *def* 式 は次の形式をもつ:

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

def 式 は、右辺の式がローカル変数やグローバル変数の値に従属する可能性はあるが相互に再帰するものではない、といったことを除けば、let 式に相等する。これは、パターン (または束縛)pb1, ..., pbn が対応する式 e1, ..., en とマッチする文脈中で、式 e の値を示す[15]。

[15]束縛が用いられている場合は、簡単に言えばパターンと一致した値はさらに第 8 章で述べられる型式または集合式によって制限を受けるということを意味する。

例題: 式の値はグローバル変数に従属するという事実に気づいてもらえるよう、*def*式 が合理的な方法で用いられる。

これは小さな例で説明することができる:

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

copy が文脈中に定義されている場所で、lib はグローバル変数である (このように lib(copy) は変数の一部の内容検索と考えることができる)。

第 13.1 節の操作 GroupRunnerUp_expl でもまた def 式の例が与えられている。

7.3 単項式または 2 項式

構文:

式 = ...
| 単項式
| 2 項式
| ... ;

単項式 = 接頭辞式
| 逆写像 ;

接頭辞式 = 単項演算子, 式 ;

単項演算子 = '+' | '-' | 'abs' | 'floor' | 'not'
| 'card' | 'power' | 'dunion' | 'dinter'
| 'hd' | 'tl' | 'len' | 'elems' | 'inds' | 'conc'
| 'dom' | 'rng' | 'merge' ;

逆写像 = 'inverse', 式 ;

2 項式 = 式, 2 項演算子, 式 ;

```
2項演算子 = '+' | '-' | '*' | '/'
          | 'rem' | 'div' | 'mod' | '**'
          | 'union' | 'inter' | '\' | 'subset'
          | 'psubset' | 'in set' | 'not in set'
          | '^'
          | '++' | 'munion' | '<:' | '<-:' | ':>' | ':->'
          | 'and' | 'or'
          | '=>' | '<=>' | '=' | '<>'
          | '<' | '<=' | '>' | '>='
          | 'comp' ;
```

意味定義: 単項式と2項式は、特定の型の値を記述する演算子と演算対象の結合である。これらすべての演算子のシグネチャについては、すでに第 4 節で述べてあるのでそれ以上の説明はここでは行わない。逆写像単項演算子は、数学的構文における接尾辞記号で記述されるため、別に取り扱う。

例題: これらの演算子を用いた例題は第 4 節で与えられるため、ここでは触れない。

7.4 条件式

構文:

```
式 = ...
   | if式
   | cases式
   | ... ;

if式 = 'if', 式, 'then', 式,
       { elseif式 }, 'else', 式 ;

elseif式 = 'elseif', 式, 'then', 式 ;

cases式 = 'cases', 式, ':',
          cases式選択肢群,
          [ ',', others式 ], 'end' ;

cases式選択肢群 = cases式選択肢,
                  { ',', cases式選択肢 } ;
```

cases 式選択肢 = パターンリスト, '->', 式 ;

others 式 = 'others', '->', 式 ;

意味定義: *if* 式 と *cases* 式は、1 つの特定の式の値を基に、複数の中から 1 つの式を選ぶことを可能にする。

if 式 は次の形式をもつ:

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

ここで e1 はブール式であり、一方 e2 と e3 はどのような型であってもよい。もし e1 が与えられた文脈中で true であるならば、 if 式は与えられた文脈中で評価された e2 の値を表す。そうでなければ、if 式は与えられた文脈上で e3 の値を表す。elseif 式の使用は、ある式の else 部分においてネストされた if-then-else 式を単に省略したものである。

cases 式 は次の形式をもつ

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

ここで e は 1 つの任意の型の式であり、pij で表すすべては 1 つ 1 つが式 e にマッチするパターンである。ei で表すのは任意の型の式であり、キーワードの others とそれに対応する式 emplus1 とはオプションとなる。cases 式では、 pij パターンの 1 つが e にマッチした文脈中で評価された ei 式の値を示す。選択された ei は、パターンの 1 つを式 e とマッチさせることができた最初の入口である。もしパターンのうちのどれも e にマッチしない場合には、 others 節がなくてはならないし、そこで cases 式は与えられた文脈中で評価される emplus1 の値を示す。

例題: VDM-SL における if 式は、大部分のプログラム言語において用いられているものに相等するが、その一方 VDM-SL における cases 式は、大部分のプログラム言語よりもより一般的なものとなる。このことは実際にパターンマッチングがおきる事例から見て取れるであろうが、しかしまた大部分のプログラム言語におけるようなパターンが定数である必要がないためでもある。条件式の使用例はマージソートアルゴリズムの記述により提供される:

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

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

cases 式によって提供されたパターンマッチングは、型の合併を扱うことに役立つ。たとえば、30 ページからの型定義 Expr を用いることで次を得る:

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

```
                                        ^print_Expr(c)^","
                                        ^print_Expr(a)
  end;

print_Const : Const -> seq1 of char
print_Const(mk_Const(c)) ==
  if is_nat(c)
  then "nat"
  else -- must be bool
       "bool";
```

関数 print_Op は同様に定義されるであろう。

7.5 限量式

構文:

式 = ...
| 限量式
| ... ;

限量式 = 全称限量式
| 存在限量式
| 1 存在限量式 ;

全称限量式 = 'forall', 束縛リスト, '&', 式 ;

存在限量式 = 'exists', 束縛リスト, '&', 式 ;

束縛リスト = 多重束縛, { ',', 多重束縛 } ;

1 存在限量式 = 'exists1', 束縛, '&', 式 ;

意味定義: 限量式には 3 つの形式がある: 全称 (forall と記述される), 存在 (exists と記述される), そして *1* 存在 (exists1 と記述される) である。以下に述べられるように、各々はブール値である true または false の値をとる。

全称限量式 は次の形式をもつ:

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

ここで各々の mbdi は多重束縛 pi in set s (あるいは型束縛であるならば pi : 型) であり、 e は mbdi のパターン識別子を含むブール式である。この値は、e を mbd1, mbd2, ..., mbdn における束縛のすべてにおいて文脈上で評価して、true であるならば true となりそうでない場合は false となる。

存在限量式 は次の形式をもつ:

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

ここで mbdi および e は、全称限量式におけるものと同じである。ここで mbd1, mbd2, ..., mbdn における束縛の少なくとも 1 つを選択した文脈上で評価した場合に e が true であったならば、この値は true となりそうでない場合は false となる。

1 存在限量式 は次の形式をもつ:

```
exists1 bd & e
```

ここで bd は 集合束縛か型束縛であり、 e は bd のパターン識別子を含むブール式である。束縛のうちのちょうど 1 つを選択した文脈上で評価して e が true であるならば、この値は true となりそうでない場合は false となる。

すべての限量式は、可能な優先度の中で最も低い優先度を持つ。これは、可能な限り長い構成式が使われることを意味する。式は、構文的に可能な限りの右側へ続く。

例題: 存在限量の例は以下の QualificationOk で提示される関数で与えられる。この関数は、[3] における化学プラント警報システムの仕様書からとってきたものであるが、ある専門家の集団が要求された資質を満たすか否かを照合するものである。

```
types

ExpertId = token;
Expert :: expertid : ExpertId
```

```
            quali : set of Qualification
inv ex == ex.quali <> ;
Qualification = <Elec> | <Mech> | <Bio> | <Chem>

functions

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

この関数 min は全称限量の例を示す:

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

1 存在限量は、すべての写像 m が満足する関数プロパティを述べるために用いることができる:

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

7.6 iota 式

構文:

式 = ...
| iota 式
| ... ;

iota 式 = 'iota', 束縛, '&', 式 ;

意味定義: *iota* 式は次の形式をもつ:

```
iota bd & e
```

ここで bd は集合束縛かまたは型束縛であり、 e は bd のパターン識別子を含むブール式である。束縛に一致して本体式 e を true とする唯一の値が存在するならば、iota 演算子を唯一用いることができる (i.e. exists1 bd & e は true でなくてはならない)。iota 式の意味定義は、本体式 (e) を満たす唯一の値を返すということである。

例題: 次に定義された値 sc1,...,sc4 を用いる

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

これより

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

最後の例は実行不可能であり、加えて最後の 2 式は未定義となることに注意しよう奸酎 O 者は式を満たす値が多くなるからであり、後者は式を満たす値がないからである。

7.7 集合式

構文:

式 = ...
| 集合列挙
| 集合内包
| 集合範囲式
| ... ;

集合列挙 = '{', [式リスト], '}' ;

式リスト = 式, { ',', 式 } ;

集合内包 = '{', 式, '|', 束縛リスト, ['&', 式], '}' ;

集合範囲式 = '{', 式, ',', '...', ',', 式, '}' ;

意味定義: 集合列挙は次の形式をもつ:

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

ここで e1 から en までは一般の式である。列挙された式の値の集合を構成する。空集合は {} と書かれなければならない。

集合内包式は次の形式をもつ:

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

述語 P が true と評価される束縛すべてのもとで、式 e を評価することで 1 つの集合が構成される。多重束縛には集合束縛と型束縛の両方を含めることができる。したがって mbdn は pat1 in set s1, pat2 : tp1, ...in set s2 というようになるであろうが、ここにおける pati はパターンであり (通常は単なる識別子である)、s1 や s2 は式で構成される集合である (これに対して tp1 は、型束縛もまた用いることができることを示すために使われている)。ただし型束縛はインタープリタでは実行できないので注意したい。

集合範囲式 は集合内包の特別な場合である。これは次の形式をもつ

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

ここでの e1 と e2 は数式である。この集合範囲式は e1 から e2 までに含まれる整数の集合を表記する。e2 が e1 よりも小さい場合には、集合範囲式は空集合を表す。

例題: Europe={<France>,<England>,<Denmark>,<Spain>} および GroupC = {sc1,sc2,sc3,sc4} (ここでの sc1,...,sc4 は前述の例にて定義されたもの) の値を用いて次を得る

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

7.8 列式

構文:

式 = ...
| 列列挙
| 列内包
| 部分列
| ... ;

列列挙 = '[', [式リスト], ']' ;

列内包 = '[', 式, '|', 集合束縛,
['&', 式], ']' ;

部分列 = 式,
'(', 式, ',', '...', ',',
式, ')' ;

意味定義: 列列挙は次の形式をもつ:

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

ここでの e1 から en は一般の式である。これは列挙された要素の列を構成する。空列は [] と書かれなければならない。

列内包 は次の形式をもつ:

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

ここでの式 e は、パターン pat からもってきた識別子を用いることになる (通常このパターンは単なる識別子となるが、唯一実際上の必要条件としては、ちょうど 1 つのパターン識別子のみがパターン中に存在するということである)。S は値 (通常は自然数) の集合である。このパターン識別子の束縛は何らかの種類の数値に対するものでなければならず、これにより結果列における要素の順を指示するために用いられる。述語 P が true と評価されるすべての束縛上で式 e を評価することにより、列を構成する。

列 l の部分列 というのは l の連続する要素からなる列; 索引 n1 以上 n2 以下のもの、である。次の形式をもつ:

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

ここでの n1 と n2 は正の整数式である。下限の n1(空でない列での最初の索引) が 1 より小さい場合は、列式は列の最初の要素から始まることとなる。上限の n2(空でない列で索引中最大のもの) が列の長さよりも大きい場合は、列式は列の最後の要素で終わることとなる。

例題: GroupA が次の列に等しい場合

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

以下が導かれる:

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

7.9 写像式

構文:

式 = ...
| 写像列挙
| 写像内包
| ... ;

写像列挙 = '{', 写, { ',', 写 }, '}'
| '{', '|->', '}' ;

写 = 式, '|->', 式 ;

写像内包 = '{', 写, '|', 束縛リスト,
['&', 式], '}' ;

意味定義: 写像列挙 は次の形式をもつ:

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

ここですべての定義域式 di と値域式 ri は一般の式である。空写像は {|->} と書かれなければならない。

写像内包 は次の形式をもつ:

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

ここでの構成 mbd1, ..., mbdn は、式 ed および er から集合 (または型) をきめる変数の多重束縛である。写像内包 は、述語 P を true と評価するすべての可能なかぎりの束縛上で、式 ed および er を評価することにより写像を構成する。

例題: GroupG は次の写像と等しいと仮定する

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

この場合に次が成り立つ:

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

7.10 組構成子式

構文:

式 = ...
| 組構成子
| ... ;

組構成子 = 'mk_', '(', 式, ',', 式リスト, ')' ;

意味定義: 組構成子式 は次の形式をとる:

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

ここで ei は一般の式である。相等および不等演算子のみが使用できる。

例題: 前述の例で定義された写像 GroupG を用いて、次が得られる:

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

7.11 レコード式

構文:

式 = ...
| レコード構成子
| レコード修正子
| ... ;

レコード構成子 = 'mk_', 名称, '(', [式リスト], ')' ;

レコード修正子 = 'mu', '(', 式, ',', レコード修正,
{ ',', レコード修正 }')' ;

レコード修正 = 識別子, '|->', 式 ;

意味定義: レコード構成子は次の形式をもつ:

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

ここでの式 (e1, e2, ..., en) の型は、レコード型 T にある対応する入り口の型に一致する。

レコード修正 は次の形式をとる:

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

ここで式 e の評価として、修正されるべきレコード値を返す。識別子 idi は、e のレコード型の中ですべて異なる名称をもつ入り口でなければならない。

例題: sc が値 mk_Score(<France>,3,0,0,9) であるならば

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

さらなる例題として関数 win の説明を行う。この関数は 2 つのチームと 1 つのスコアをもつ。スコアの集合から与えられているチームに相当するスコア (勝ったチームには wsc、負けたチームには lsc) を各々割り当て、mu 演算子を用いてこれらを更新する。チームの集合はここで、新しいスコアをもとのものと置き換えることで更新される。

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

7.12 適用式

構文:

式 = ...
| 適用
| 項目選択
| 組選択
| 関数型インスタンス化
| ... ;

適用 = 式, '(', [式リスト], ')' ;

項目選択 = 式, '.', 識別子 ;

組選択 = 式, '.#', 数字 ;

関数型インスタンス化 = 名称, '[', 型, { ',', 型 }, ']' ;

意味定義: 項目選択式 はレコードに対して用いることができるが、第 4.2.5 節ですでに説明したのでここではそれ以上の説明は行わない。適用 は、ある写像において検索を行い、列に索引をし、最後に関数を呼び出すために用いられる。第 4.2.3 節で、写像において検索を行うとはどういうことかはすでに述べてある。同様に第 4.2.2 節では、列に索引をするとはどのように行うのかが説明されている。

VDM-SL においては、ここで更に 1 つの操作を呼び出すことが可能である。これは標準 VDM-SL においては許されていないことであり、この種の操作呼び出しは状態を変更してしまう可能性があるので、混合式においては慎重に使用されるべきである。このような操作呼び出しで例外を起こすことが許されてはいないことに注意したい。

このような操作呼出しでは評価の順が重要となる可能性がある。したがって型検査では、式の中でユーザーが操作呼出しを有効化や無効化することを許す。

組選択式は、組から特別な構成要素を抽出するために用いられる。式の意味は、 もし `e` がいくつかの組 `mk_(v1,...,vN)` であると評価され、 `M` が範囲 `{1,...,N}` 内の 1 つの整数であるならば、 `e.#M` は `vM` となるということである。`M` が `{1,...,N}` からはずれているならば、この式は未定義である。

関数型インスタンス化 は、適当な型をもつ多相関数のインスタンス生成に用いられる。これは次の形式をもつ:

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

ここで `pf` は多相関数の名称であり、 `t1, ..., tn` は型である。結果の関数は、関数定義で与えられた変数型の名称の代わりに、型 `t1, ..., tn` を用いる。

例題: GroupA は 1 つの列 (56 ページ)、 GroupG は 1 つの写像 (58 ページ)、そして selection_sort は 1 つの関数 (44 ページ) であったことを思い起こそう:

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

多相関数使用と関数型インスタンス化の 1 つの例として、第 6 節から例題の関数を用いる:

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

≡

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

7.13 ラムダ式

構文:

式 = ...
| ラムダ式
| ... ;

ラムダ式 = 'lambda', 型束縛リスト, '&', 式 ;

型束縛リスト = 型束縛, { ',', 型束縛 } ;

型束縛 = パターン, ':', 型 ;

意味定義: ラムダ式 は次の形式をもつ:

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

ここで pati はパターン、Ti は型式、そして e は本体式である。パターン pati におけるパターン識別子のスコープが本体式である。ラムダ式は多相ではありえないが、それとは別に、意味定義においては第 6 節に説明される陽関数定義に相当する。ラムダ式によって定義される関数は、入れ子になった本体中で新しいラムダ式を用いることでカリー化することが可能となる。ラムダ式が 1 つの識別子と結びついたとき、再帰関数を定義することもまた可能である。

例題: 以下のようにラムダ式を用いて、増加関数を定義することができる:

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

さらに加算関数はカリー化できる:

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

もしこれが唯一の引数に適用された場合には、新しいラムダ式が返される:

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

ラムダ式は、高階関数との関連で用いられる場合に役立つ。たとえば 43 ページに定義される関数 set_filter を用いてみると:

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

7.14 narrow 式

構文:

式 = ...
| 式 ;

narrow 式 = 'narrow_', '(', 式, ',', 型, ')' ;

意味定義: *narrow*式の値は、与えた式の結果の型を、指定された型に変換したものである。

静的型チェックおよび動的型チェックにより、無関係の型間の変換は型エラーとなる。

例題 1: この例では、Test() および Test'() の実行結果には差がないが、Test() は「def」型チェックエラーとなる。

```
class A

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

operations
public
Test: () ==> nat
Test() ==
 let s : S = mk_C1(1)
 in
   let c : C1 = s
   in
     return c.a;

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

7.15 is 式

構文:

式 = ...
| 一般 is 式
| ... ;

一般 is 式 = is 式
| 型判定 ;

is 式 = 'is_', 名称, '(', 式, ')'
| is 基本型, '(', 式, ')' ;

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

型判定 = 'is_', '(', 式, ',', 型, ')' ;

意味定義: *is* 式 は基本値かまたはレコード値 (なにかのレコード型に属するタグ付けされた値) とともに用いられる。この is 式は、与えられた値が指定された基本型に属する場合、または値が指定されたタグを持つ場合に、true となる。他の場合は false となる。

型判定は、型が静的には決定されえない式に対して用いることができることから、より一般的な形式である。式 is_(e,t) は、 e が t 型 でありその場合にのみ、true となる。

例題: 27 ページに定義されたレコード型 Score を用いて次を得る:

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

型判定の例は以下のとおり:

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

加えて 30 にも例題が載せられている。

7.16 リテラルと名称

構文:

```
式 = ...
   | 名称
   | 旧名称
   | 記号リテラル
   | ... ;

名称 = 識別子, [ '`', 識別子 ] ;

名称リスト = 名称, { ',', 名称 } ;

旧名称 = 識別子, '~' ;
```

意味定義: 名称 と 旧名称 は、関数、操作、値、状態構成要素の定義にアクセスするためによく用いられる。名称 は次の形式をもつ:

```
id1`id2
```

ここで `id1` と `id2` は単なる識別子である。名称が唯一の識別子で構成される場合は、その識別子はスコープ内で定義されている。つまり、ローカルにパターン識別子かパターン変数として定義されているか、あるいはグローバルに現モジュール内で関数、操作、値、またはグローバル変数として定義されているか、いずれかである。そうでない場合は、識別子 `id1` がコンストラクタが定義されている場所のモジュール名を示している (第 14 節および付録 B も参照)。旧名称 は、操作定義の事後条件 (第 12 節参照) および仕様文の事後条件 (第 13.14 節参照) において、グローバル変数の旧値にアクセスするためによく用いられる。これは次の形式をもつ:

```
id~
```

ここで `id` は状態構成要素である。

記号リテラル はいくつかの基本型における定数値である。

例題: 名称 と 記号リテラル はこの本の中ですべての例題を通して用いられている (付録 B.2 参照)。

旧名称の使用の例は、以下のように定義された状態を考える:

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

変数 index を増加させる陰操作を定義することができる:

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

操作 IncIndex は、ext wr 節に示されるように、変数 index を操作する。事後条件の中で、 index の新しい値は index の旧値に 1 を足したものと等しい。(これ以上は第 12 節の操作についてを参照)。

module 名の簡単な例としては、以下のように、 build_rel という関数が CGRel という module において定義された (そしてエクスポートされた) と仮定する:

```
types

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

functions

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

別の module においては、最初にモジュール CGRell を輸入し、それから、以下の呼出しを行なうことでこの関数をアクセスすることができる

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

7.17 未定義式

構文:

式 = ...
| 未定義式 ;

未定義式 = 'undefined' ;

意味定義: 未定義式 は、ある式の結果が定義されないことを明白に述べるために用いられる。たとえばこれは、if-then-else 式で else 分岐を評価した結果をどうすべきかが決定されていない場合などに、用いることができるであろう。未定義式 が評価される場合、インタプリタは実行を終了し未定義式が評価されたと記録する。

実用において、未定義式の使用は事前条件のとは異なる: 事前条件の使用とは、関数が呼ばれたときに事前条件が満たされることを保障するのは呼び出す側の責任であることを意味する; 未定義式の使用であれば、エラー処理を行うのは呼び出された関数の責任となる。

例題: Score 値の build の前に、型の不変条件が保たれるかをチェックすることができる:

```
build_score : Team * nat * nat * nat * nat -> Score
build_score (t,w,d,l,p) ==
  if 3 * w + d = p
  then mk_Score(t,w,d,l,p)
  else undefined
```

7.18 事前条件式

構文:

式 = ...
| 事前条件式 ;

事前条件式 = 'pre_', '(', 式,
[{ ',', 式 }], ')' ;

意味定義: e が関数型であると仮定すると、 式 pre_(e,e1,...,en) は、e の事前条件が引数 e1,...,em に対して true でありかつその場合にのみ true となるが、ここで m は e の事前条件の arity(引数の数) である。e が関数でなかったり、 m > n であったりした場合は、結果は true となる。e が事前条件をもたない場合は、式は true と等しい。

例題: 以下のように定義された関数 f と g を考えよう

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

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

この場合、次の式は

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

以下と等しくなる

- h が f に束縛されている場合は pre_f(1,0) と等しいと考えられるため、false となる;
- h が g に束縛されている場合は pre_g(1) と等しいと考えられるため、true となる;

- h が lambda mk_(x,y):nat * nat & x div y に束縛されている場合はこの関数に対して定義された事前条件がないため、 true となる。

h がいかに束縛されていたとしても、最後の引数 (-1) は決して使われないことに注意しよう。

8 パターン

構文: パターン束縛 = パターン | 束縛 ;

パターン = パターン識別子
| 一致値
| 集合列挙パターン
| 集合合併パターン
| 列列挙パターン
| 列連結パターン
| 写像列挙パターン
| 写像併合パターン
| 組パターン
| レコードパターン ;

パターン識別子 = 識別子 | '-' ;

一致値 = 記号リテラル
| '(', 式, ')' ;

集合列挙パターン = '{', [パターンリスト], '}' ;

集合合併パターン = パターン, 'union', パターン ;

列列挙パターン = '[', [パターンリスト], ']' ;

列連結パターン = パターン, '^', パターン ;

写像列挙パターン = '{', [写パターンリスト], '}' ;

写パターンリスト = 写パターン, { ',', 写パターン } ;

写パターン = パターン, '|->', パターン ;

写像併合パターン = パターン, 'munion', パターン ;

組パターン = 'mk_(', パターン, ',', パターンリスト, ')' ;

レコードパターン = 'mk_', 名称, '(', [パターンリスト], ')' ;

パターンリスト = パターン, { ',', パターン } ;

意味定義: パターンは常に文脈中で用いられ、1 つの特定の型の 1 つの値に一致する。マッチングでは、あるパターンがある値と一致する可能性があるかの照合を行い、そしてパターン中のパターン識別子に対応する値を結びつけ、識別子がそのスコープ内で、これらの値を意味するようにする。パターンを用いることのできるいくつかの場合においては、束縛も同様に用いることができる (次節を参照)。もし束縛が用いられていたら、それは単純に言って、与えられたパターンに一致する可能性のある値を束縛することに更なる情報 (型式または集合式) が用いられていることを意味する。

マッチングは次のように定義される

1. パターン識別子 はどんな型にも合致するしどんな値にも一致し得る。それが識別子であるならば、その識別子はその値に束縛される; それが don't-care 記号 '-' であるならば、どのような束縛も起こらない。
2. 一致値 はそれ自身の値に対してのみ一致し得る; どのような束縛もなされない。一致値がたとえば 7 とか <RED>とかのようにリテラルでない場合は、パターン識別子に対してこれを区別するために、括弧にかこまれた式でなければならない。
3. 集合列挙パターン は集合値のみと適合する。1 つ 1 つのパターンは 1 つの集合の異なる要素と一致させられ; すべての要素が一致しなければならない。
4. 集合合併パターン は集合値のみと適合する。1 つの集合を 2 つに分けた部分集合に対して、2 つのパターンが一致する。2 つの部分集合は、互いに素で、かつ合併すると元の集合になるように選ばれる。元の集合の要素数が 2 以上の時、2 つの部分集合は空で無いように分割される。元の集合の要素数が 1 の時は、1 つの部分集合は、空となる。

5. 列列挙パターン は唯一列値にのみ合致する。各々のパターンは列値中の対応する要素に対して一致する; 列長とパターン数は等しくなければならない。
6. 列連結パターン は唯一列値とのみ合致する。2 つのパターンは、共に連結するともとの列値をつくることができる 2 つの部分列に、一致する。2 つの部分列は常に空でないように選ばれる。
7. 写像列挙パターン は写像値とのみ合致する。

8. 写パターンリスト は 1 つの写像の、それぞれ異なる写（maplet）と一致する; すべての写が一致しなければならない。
9. 写像併合パターン は写像値とのみ合致する。1 つの写像を 2 つに分けた部分写像に対して、2 つのパターンが一致する。2 つの部分写像は、互いに素となり、かつ併合すると元の写像と一致するように選ばれる。元の写像の要素数が 2 以上の時、2 つの部分写像は空で無いように分割される。元の写像の要素数が 1 の時は、1 つの部分写像は、空となる。
10. 組パターン は同じ要素数をもつ組にのみ合致する。パターンの各々は、組値の中で対応する要素に対して一致させられる。
11. レコードパターン は同じタグをもつレコード値にのみ適合する。パターンの各々は、レコード値の項目に対して一致させられる。レコードのすべての項目が一致させられなくてはならない。

例題: 以下にパターンの使用例を説明する。

パターン識別子の例 最も単純なパターンはパターン識別子である。この例は次に述べる let 式で与えられる:

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

ここで識別子 `top` は列 `GroupA` の先頭と結びつき、したがって識別子は let 式の本体で用いられることが許される。

一致値の例 以下の例では一致値を用いる:

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

一致値は唯一それ自身の値と一致させることが可能なので、ここで GroupA の先頭のチームが <Brazil> であるならば最初の節で一致する; もし GroupA の先頭のチームが<France> であるなら 2 番目の節で一致する。これら以外は others が一致する。ここで a を囲んだ括弧の使用が、a を一致値とみなすよう強要していることに留意しよう。

集合列挙パターンの例 集合列挙は、パターンを 1 つの集合の要素と一致させる。たとえば次において

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

識別子 sc1, sc2, sc3 および sc4 は GroupA の 4 つの要素と結び付けられる。束縛の選択はゆるいものであることに注目しよう - たとえば sc1 は elemsGroupA の [どのような] 要素と結び付いてもよい。この場合、もし elems GroupA がちょうど 4 つの要素を含んでいるわけではなかったら、この式は良形とはいえない。

集合合併パターンの例 集合合併パターンは、集合を再帰関数呼出しに分解させるために用いることができる。この 1 つの例は集合を (任意の順での) 列に変換する関数 set2seq である:

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

case の 3 番目の選択肢で、集合合併パターンを使用しているのがわかる。これは s1 と s2 を s の任意の部分集合に束縛し、それによって s を区分けする。Toolbox インタープリタは常に互いに素の区分けを実現する。

列列挙パターンの例 列列挙パターンは、1 つの列から指定された要素を抽出するために用いることができる。この 1 つの例として関数 promoted があり、これはスコアの列の最初から 2 つの要素を抽出し、チームの中の対応する 2 つを返す:

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

ここで sc1 は引数列の先頭と結びつき、sc2 は列の 2 番目の要素と結びつく。もし promoted が要素数が 2 つない列で呼び出されるなら、ランタイムエラーが起きる。リストの残りの要素には興味を持たないので、それら残りに対して don't care パターンを用いていることに注目したい。

列連結パターンの例 前に述べた例でも、列連結パターンの使用を行っている。もうひとつの例として関数 quicksort があるが、これは標準のクイックソートアルゴリズムを実装している:

```
quicksort : seq of nat -> seq of nat
quicksort (l) ==
  cases l:
    [] -> [],
    [x] -> [x],
    [x,y] -> if x < y then [x,y] else [y,x],
    -^[x]^- -> quicksort ([y | y in set elems l & y < x]) ^
               [x] ^ quicksort ([y | y in set elems l & y > x])
  end
```

ここで、 case 式の最後の cases 式選択肢で、列連結パターンは l をある任意のピボット (かなめ) 要素と 2 つの部分列に分解するのに用いられている。ピボットはリストをピボットより小さい値と大きい値に区分けするために用いられ、2 つの区分けされた部分は再帰的にソートされる。

写像列挙パターンの例 写像列挙パターンは、パターンを 1 つの写像と個々の写（maplet）と一致させる。例えば、次の例では

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

a は、対応する定義域の値が 1 なので、4 と一致する。a が 4 なので、定義域の値が 4 である写の値域の値すなわち b の値は 2 になる。同様に、b が 2 なので、c は 3 になる。

写像併合パターンの例 写像併合パターンは、写像の写（maplet）を 1 つずつ処理する再帰関数に用いることができる。ここでは、写像を（任意の順番で）写の列に変換する関数 map2seq を示す。

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

ここで、case 式の 3 番目の cases 式選択肢で、写像併合パターンを使用している。m1 と m2 を写像 m の任意の（互いに素な）部分写像に束縛する。

組パターンの例 組パターンは、組構成要素を識別子と結びつけるために用いることができる。たとえば上で定義された関数 promoted は 2 つを返すので、以下の値定義では GroupA の勝ったチームの方を識別子 Awinner に結びつける:

```
  values

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

レコードパターンの例 レコードパターンはレコードのいくつかの項目が同じ式で用いられるときに役立つ。たとえば次の式は、チーム名から点数スコアへの写像を構成する:

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

49 ページの関数 print_Expr もまた、レコードパターンのいくつかの例を与えてくれる。

9 束縛

構文:

束縛 = 集合束縛 | 型束縛 ;

集合束縛 = パターン, 'in set', 式 ;

型束縛 = パターン, ':', 型 ;

束縛リスト = 多重束縛, { ',', 多重束縛 } ;

多重束縛 = 多重集合束縛
| 多重型束縛 ;

多重集合束縛 = パターンリスト, 'in set', 式 ;

多重型束縛 = パターンリスト, ':', 型 ;

意味定義: 束縛は、あるパターンをある値に一致させる。集合束縛 において、値は束縛の集合式によって定義された集合から選ばれる。型束縛 において、値は型式で定義された型から選ばれる。多重束縛は、いくつかのパターンが同じ集合または型に束縛されることを除けば 束縛 と同じである。型束縛はインタープリタで実行させることは **できない** ことに注意しよう。これは、インタープリタに自然数というような無限の定義域の検索を要求するということであるからだ。

例題: 束縛は主に、これらの例にみられるように限量式や内包で用いられる:

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

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

{ y | y: nat & y > 3 }

occurs : seq1 of char * seq1 of char -> bool
occurs (substr,str) ==
  exists i,j in set inds str & substr = str(i,...,j);
```

10 値 (定数) 定義

VDM-SL では定数値の定義をサポートする。値定義については、伝統的プログラム言語における定数定義に相当する。

構文: 値定義群 = 'value', [値定義, { ';', 値定義 }, [';']] ;

値定義 = パターン, [':', 型], '=', 式 ;

意味定義: 値定義は次の形式をもつ:

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

グローバル値 (値定義で定義された) は、 VDM-SL 記述の全水準で参照が可能である。しかしながら、1 つの記述が実行可能であるためには、値定義の列に用いられる前段階でそれらの値は定義されていなければならない。この "使用前の宣言" 原則は、インタープリタが値定義にのみ使う。このことはたとえば、関数などは宣言される前に用いられることが可能なのであ

る。標準 VDM-SL では、定義の順番における制限はいっさい存在しない。同様に型制限を提供することが可能であるから、このことでより正確な型情報を得るのに役立てることができる。

例題: 以下の例は [3] からとったものであるが、識別子 p1 と eid2 にトークン値を与え、e3 に Expert レコード値を与え、そして a1 に Alarm レコード値を与える。

```
types

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

values

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

この例が示すように、ある値はそれ自身が定義される前に定義された他の値に依存できる。トップレベル仕様記述は多くのファイルやモジュールからの仕様記述から成り立つことができる (節 14 参照)。命令が重要であるように、他のモジュール内で定義された値に依存させないことは、良い慣習である。

11 状態定義

グローバル変数が仕様記述で要求されれば、状態定義をすることは可能である。状態定義のコンポーネントは操作内で参照できるグローバル変数のコレクション

と見なすことが可能である。モジュールの状態はモジュール内の (その状態を使用している) 操作定義のいずれかがインタープリタによって使用される前に初期化される。

構文: 状態定義 = 'state', 識別子, 'of', 項目リスト,
[不変条件], [初期化], 'end', [';'] ;

不変条件 = 'inv', 不変条件初期関数 ;

初期化 = 'init', 不変条件初期関数 ;

不変条件初期関数 = パターン, '==', 式 ;

意味定義: 状態定義は次の形式をもつ:

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

状態識別子 `idn` は特定の型 `typen` であると宣言される。不変条件 `inv` はいつも状態 `ident` が保たなければならない特性を意味している論理式である。`init` は初めに成立しなければならない状態を指示する。インタープリタを使用するためには、(もしも、状態を使用している操作が実行されるならば) 初期化述語が必要であることに注意すべきである。加えてこの初期化述語の本体は状態全体の名称 (その名称はパターンとして使用されなくてはならない) を等式の左辺値とし、右辺値は正しい型のレコード値として評価される二項相当式でなくてはならない。これはインタープリタが `init` 状態を評価すること可能にする。初期化述語の簡単な例を以下に示す:

```
state St of
  x:nat
  y:nat
```

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

不変条件と初期値の両方の仕様記述では、全体として状態を操作しなければならない。また、それを状態名 (例を参照) でタグ付けされたレコードとして参照することによって行う。状態中の項目が operation で操作されるとき、状態名が前に付いていない項目名によって、項目は直接参照されなければならない。

例: 以下の例では、一つの状態変数を作成している:

```
types

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

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

functions

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

不変条件において、各グループには 4 つのチームがあり、どのチームも 3 ゲーム以上行わないことを提示する。初めは、どのチームもゲームをしていない。

12 操作定義

操作については第 5 節にすでに述べてきた。一般的な形式はここで述べる。

構文:

操作定義群 = 'operations', [操作定義,
{ ';', 操作定義 }, [';']] ;

操作定義 = 陽操作定義
| 陰操作定義
| 拡張陽操作定義 ;

陽操作定義 = 識別子, ':', 操作型,
識別子, パラメーター群,
'==',
操作本体,
['pre', 式],
['post', 式] ;

陰操作定義 = 識別子, パラメーター型,
[識別子型ペアリスト],
陰操作本体 ;

陰操作本体 = [外部節],
['pre', 式],
'post', 式,
[例外] ;

拡張陽操作定義 = 識別子,
パラメーター型,
[識別子型ペアリスト],
'==', 操作本体,
[外部節],

[‘pre’, 式],
[‘post’, 式],
[例外] ;

操作型 = 任意の型, ‘==>’, 任意の型 ;

任意の型 = 型 | ‘()’ ;

パラメーター群 = ‘(’, [パターンリスト], ‘)’ ;

パターンリスト = パターン, { ‘,’, パターン } ;

操作本体 = 文
| ‘is not yet specified’ ;

外部節 = ‘ext’, var 情報, { var 情報 } ;

var 情報 = モード, 名称リスト, [‘:’, 型] ;

モード = ‘rd’ | ‘wr’ ;

名称リスト = 識別子, { ‘,’, 識別子 } ;

例外 = ‘errs’, エラーリスト ;

エラーリスト = エラー, { エラー } ;

エラー = 識別子, ‘:’, 式, ‘->’, 式 ;

意味定義: 陽操作の以下の例は、1 つのチームがもうひとつを打ち負かす場合に状態 GroupPhase を更新する。

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

```
                                else if sc.team = lt
                                then mu(sc, lost |-> sc.lost + 1)
                                else sc
                          | sc in set gps(gp)}}
    pre  exists gp in set dom gps &
           {wt,lt} subset {sc.team | sc in set gps(gp)};
```

1 つの陽操作は 1 つの文 (あるいは 1 つのブロック文を用いてまとめられたいくつかの文) からなるが、これについては第 13 節で述べられている。文は、必要とするどのような状態変数に対しても、適当と判断したときに読込みや書出しを行いアクセスすることが許されている。

陰操作は、オプションである事前条件、または必要不可欠な事後条件を用いて指定される。たとえば、ここに暗黙に陰操作 Win を指定できる:

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

外部節の項目は、操作が扱うはずの状態変数をリストする。予約語である rd の後にリストされた状態変数は読み取りのみができるが、その一方 wr の後にリストされた変数は読みとり書きだしの両方行うことができる。

これらの事前、事後条件に対して、インタープリタも操作定義の事前、事後条件として新しい関数を作成する。しかし、仕様記述がグローバルな状態を含んでいるならば、状態は新たに作成された関数の一部である。した

がって、以下のシグネチャを持つ関数が、事前、事後条件をもった操作に対して作成される[16]:

```
pre_Op : InType * State +> bool

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

以下は例外である:

- もし、操作が引数を取らないなら、シグネチャの InType 部分を pre_Op と post_Op の両方で省略する。
- もし、操作が値を返さないなら、OutType 部分を post_Op シグネチャにおいて省略する。
- もし、仕様記述が状態を定義しないなら、両方のシグネチャの State 部分は省略する。

post_Op シグネチャにおいて、最初の State 部分は旧状態であるに対して、2 番目の State 部分は操作呼び出し後の状態である。

例えば、以下の仕様記述を考える:

[16] しかしながら、これらの事前、事後条件の述語は単にブール関数であることを覚えておくべきである。また、その状態のコンポーネントは、このような述語を呼び出すことによって変更してはいけない

```
module A

definitions

state St of
  n : nat
end

operations

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

Op2 () b : nat
post b = 2;

Op3 ()
post true

end A
```

```
module B

definitions

operations

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

Op2 () b : nat
post b = 2;

Op3 ()
post true

end B
```

module A に対しこの仕様記述で定義されている事前事後条件の引用方法を以下に示す

引用式	説明
pre_Op1(1,mk_St(2))	n を 2 に、a を 1 に束縛する
post_Op1(1,2,mk_St(1), mk_St(2))	a を 1 に、b を 2 に、n の前の状態を 1 に、n の後の状態を 2 に束縛する
post_Op2(2,mk_St(1), mk_St(2))	b を 2 に、n の前の状態を 1 に、n の後の状態を 2 に束縛する
post_Op3(mk_St(1), mk_St(2))	n の前の状態を 1 に、n の後の状態を 2 に束縛する

module B に対しこの仕様記述で定義されている事前事後条件の引用方法を以下に示す

引用式	説明
pre_Op1(1)	a を 1 に束縛する
post_Op1(1,2)	a を 1 に、b を 2 に束縛する
post_Op2(2)	b を 2 に束縛する
post_Op3()	何も束縛しない

例外節は、ある操作がエラー状態にどのように対処するかを記述をすることに用いることができる。理論的根拠を持つための例外節は、正常なケースと例外ケースを切り離す分離することを可能にする。例外を用いた記述では、どのように例外を起こすかについて言及はしていないが、どのような環境でエラー状態が起こり得るか、また操作を呼び出した結果どのような影響が起きるかについて示す手段が与えられる。

例外節は次の形式をもつ:

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

条件名 COND1, . . . , CONDn は識別子で、起きる可能性があるエラーの種類を記述する[17]。条件式 c1, . . . , cn は、異なる種類のエラーに対する事前条件と考えることができる。このようにこれらの式においては、引数リストから識別子をまた外部節リストからは変数を用いることができる (それらは事前条件と同じスコープをもつ)。結果の式である r1,. . . , rn は相対的にみれば、異なる種類のエラーに対する事後条件と同じと考えられる。これらの式においては、結果の識別子とグローバル変数 (書き込みのできる) の旧値もまた用いることができる。このように、ここでのスコープは事後条件のスコープに相当する。

表面的には、ここでの例外と事前条件との間にはいくらか重複があるようにみえる。しかしながらこれらの間には、どちらをいつ用いるべきか指し示す概念的な違いが存在する。事前条件は、その操作の呼び出しが正しく行なわれるためにどんなことを保証しなければならないかを指定する; 例外節は、例外条件が満たされたときに記述された操作がエラー処理の責任をとることを示す。したがって通常は、例外節と事前条件は重複しない。

操作における次の例は、以下の状態定義を使用する:

[17] これらの名前は単に記憶を助けてくれる値であり、つまり意味定義上は重要でない。

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

この例では、陰操作の例外がどのように用いられるか示される:

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

これはデキュー操作であって、 型 Queue の グローバル変数 q を用いて、キューから型 Elem の 要素 e をとりのぞく。ここでの例外としては、その中で例外節が操作がどのように行われるかを指定するキューが、空である場合がある。

Toolbox はここで以下の関数を作成することに注意する:

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

13 文

この章では、異なる種類の文を 1 種類ごとに記述する。各々は、次の方法で記述されていく:

- BNF 構文記法。
- 形式的でない意味定義記述。
- 使用例を 1 つ。

13.1 let 文

構文:

```
文 = let 文
  | let be 文
  | ... ;
```

let 文 = 'let', ローカル定義, { ',', ローカル定義 },
'in', 文 ;

let be 文 = 'let', 束縛, ['be', 'st', 式], 'in',
文 ;

ローカル定義 = 値定義
| 関数定義 ;

値定義 = パターン, [':', 型], '=', 式 ;

ここでの構成要素である "関数定義" は第 6 節に記述されている。

意味定義: *let* 文 と *let-be-such-that* 文 は、*in* の部分が式であるかわりに文であることを除けば、各々がそれぞれ *let* と *let-be-such-that* 式 に対応し似ている。これは以下のように説明することができる:

単純な *let* 文 は次の形式をもつ:

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

ここで `p1, ..., pn` はパターン、`e1, ..., en` は対応するパターンである `pi` に一致する式、そして `s` は任意の型の文であるが `p1, ..., pn` というパターン識別子を含む。これは、パターン `p1, ..., pn` が式 `e1, ..., en` に対して一致する文脈中での、文 `s` の評価を示す。

ローカル関数定義を用いることで、より高度な let 文をつくることもできる。これを行う意味定義は単純で、このようなローカルに定義された関数のスコープは let 文の本体に制限される。

VDM-SL では、複数の定義が互いに再帰的になるかもしれない。しかしながら、これは VDM-SL でインタプリタによってサポートされていない。その上、定義を使用されている前に、すべての構成要素が定義されるように命令しなければならない。

let-be-such-that 文は次の形式をもつ

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

ここで b は束縛であって集合値 (または型) に対するパターンからなり、 e はブール式、 そして s は文であって b におけるパターンのパターン識別子を含むものである。be st e の部分はオプションである。この式は、b からの集合 (または型) の要素に対し b からのパターンが一致する文脈中での、文 s の評価を示すものである[18]。be st 式である e が存在するとき、唯一このような束縛が、一致させる文脈中で e が真と評価される場所で用いられる。

例題: let be st 文の 1 つの例は、操作 GroupWinner 中にあり、あるグループ内での勝チームを返してくれるものである:

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

仲間の操作である GroupRunnerUp もまた同様に、簡単な let 文の例を与えてくれる:

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

ここにおける def 文 (第 13.2 節参照) の使用に注意; 右辺が操作呼出しであるために let 文ではなくこれを用いるが、したがってこれは式ではない。

[18]集合束縛のみがインタープリタで実行できるということを思い出そう。

13.2 def文

構文:

```
文 = ...
   | def文
   | ... ;

def文 = 'def', 相等定義,
        { ';', 相等定義 },[ ';' ], 'in',
        文 ;

相等定義 = パターン束縛, '=', 式 ;
```

意味定義: *def*文 は次の形式をもつ:

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

*def*文 は、右辺において操作呼出しを用いることが許される点を除けば *def* 式に相当するものである。このような、状態を変化させる操作もここで用いることができ、またもし複数の定義がなされている場合には、出現した順に評価が行われる。このことは、パターン (または束縛)pb1，...，pbn が対応する式や演算呼出し e1，...，en によって返される値と一致する文脈中の文 s の評価の外延を表す[19]。

例題: 以下の列が与えられている:

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

操作 SecondRound は組の列を返し、第 2 ラウンドゲームが def 文の例を与えていることを表わす:

[19]束縛が用いられるのであれば第 8 節で説明されるように、パターンと一致し得る値はさらに型や集合式により束縛がなされることを単純に意味する。

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

13.3 ブロック文

構文:

文 = ...
| ブロック文
| ... ;

ブロック文 = '(', { dcl 文 },
文, { ';', 文 }, [';'], ')' ;

dcl 文 = 'dcl', 代入定義,
{ ',', 代入定義 }, ';' ;

代入定義 = 識別子, ':', 型, [':=', 式] ;

意味定義: ブロック文は、伝統的な高水準プログラム言語におけるブロック文に相当する。ブロック文を用いることで、(宣言文を使用して) ブロック文の本体内での変更が許されたローカル定義変数の使用が可能となる。これは単に、個々の文が規定することを順に実行することを示すものである。文の列において 1 つの値を返す最初の文により、文の列を終了させるための評価がひき起こされる。この値はブロック文の値として返される。ブロック内のどの文も値を返さない場合には、ブロック文の評価はブロック内の最後の文が評価されたときに終了する。ブロック文が中止された場合には、ローカル変数の値は開放される。このように、これら変数のスコープは単純にブロック文の内部である。

例題: 次のような state definition

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

に関連したものとしてここに述べる Swap 操作は、変数 x と y の値交換を行うためにブロック文を用いる:

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

13.4 代入文

構文:

文 = ...
| 一般代入文
| ... ;

一般代入文 = 代入文
| 多重代入文 ;

代入文 = 状態指示子, ':=', 式 ;

状態指示子 = 名称
| 項目参照
| 写像参照または列参照 ;

項目参照 = 状態指示子, '.', 識別子 ;

写像参照または列参照 = 状態指示子, '(', 式, ')' ;

多重代入文 = 'atomic', '(' 代入文, ';',
代入文,
{ ';', 代入文 }')' ;

意味定義: 代入文 は、伝統的な高水準プログラム言語における代入文を一般化したものに相当する。これを用いてグローバル状態またはローカル状態の値を変更する。したがって代入文は、状態を変える副作用を起こす。しかしながら、単純にある状態の一部を変更することができるように、代入文の左辺を状態指定子とすることができる。状態指定子とは、単純なグローバル変数の名前、変数項目への参照、変数の写像参照、または変数の列参照、である。この方法で、ある状態の小さな構成子の値を変更することが可能だ。たとえば、もし状態構成要素が写像であったならば、写像のひとつの要素を変更することが可能である。

代入文は次の形式をもつ:

```
sd := ec
```

ここで sd は状態指定子であり、 ec は式または操作呼出しである。代入文は、(式または操作呼出しの) 右辺に記述された既に与えられている状態構成要素への変更を示す。もし右辺が操作変更の状態であるならば、その操作は代入が行われる前に (それ相当の副作用を伴い) 実行される。

多重代入もまた可能である。これは次の形式をもつ：

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

右辺の式または操作呼出しはすべて、実行されるかあるいは評価され、その結果は相当の状態指定子に結び付けられる。右辺は、不変条件評価に関しては原子的に実行される。

例題: 前に述べた例 (Swap) における操作は、通常の代入を表していた。次の Win_sd 操作は、82 ページの Win を手直ししたものであり、特定の写像キーに代入をおこなう状態指示子の仕様記述を行っている:

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

SelectionSort 操作 は 44 ページにおける selection_sort 関数の状態使用版である。これは、 92 ページに定義された state St を用いて、特定の列索引の内容を変更するための状態指示子の使用を述べている。

```
functions

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

operations

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

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

13.5 条件文

構文:

```
文 = ...
   | if文
   | cases文
   | ... ;

if文 = 'if', 式, 'then', 文,
       { elseif文 }, [ 'else', 文 ] ;

elseif文 = 'elseif', 式, 'then', 文 ;

cases文 = 'cases', 式, ':',
          cases文選択肢群,
          [ ',', others文 ], 'end' ;

cases式選択肢群 = cases文選択肢,
                  { ',', cases文選択肢 } ;

cases文選択肢 = パターンリスト, '->', 文 ;

others文 = 'others', '->', 文 ;
```

意味定義: *if* 文の意味定義は、文に代わることを除けば (また else 部分がオプションであることも除けば)、第 7.4 節に述べられた *if* 式 に相当する[20]。

cases 文 に対する意味定義は、文に代わることを除けば、第 7.4 節に述べられた *cases* 式 に相当する。

例題: 以下のシグネチャをもつ、関数 `clear_winner` と `winner_by_more_wins` そして演算 `RandomElement` を仮定する:

[20] else 部分が省略される場合は、意味定義上は else skip を用いるのと同じである。

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

その後、操作 GroupWinner_if ではネストした if 文使用の実例が挙げられている (iota 式については 52 ページで紹介されている):

```
GroupWinner_if : GroupName ==> Team
GroupWinner_if (gp) ==
  if clear_winner(gps(gp))
   -- 他のどのスコアより点数が多い gps(gp) の
   -- 唯一のスコアを返す
  then return ((iota sc in set gps(gp) &
                  forall sc' in set gps(gp) \ {sc} &
                    sc.points > sc'.points).team)
  else if winner_by_more_wins(gps(gp))
   -- 最大の点数であり最大点数の他のスコアより多く勝っている
   -- gps(gp) における唯一のスコアを返す
  then return ((iota sc in set gps(gp) &
              forall sc' in set gps(gp) f {sc} &
                (sc.points > sc'.points) or
                (sc.points = sc'.points and
                 sc.won > sc'.won)).team)
   -- まったくの勝者はなく、よって最高スコアの中から
   -- 無作為にスコアを選ぶ
  else RandomElement ( {sc.team | sc in set gps(gp) &
                          forall sc' in set gps(gp) &
                           sc'.points <= sc.points} );
```

代わりとして、この操作に対しては一致値パターンをもつ cases 文を用いることができるだろう:

```
GroupWinner_cases : GroupName ==> Team
GroupWinner_cases (gp) ==
  cases true:
```

```
  (clear_winner(gps(gp))) ->
       return ((iota sc in set gps(gp) &
                     forall sc' in set gps(gp) \ {sc} &
                      sc.points > sc'.points).team),

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

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

13.6 for ループ文

構文:

文 = ...
| 列 for ループ
| 集合 for ループ
| 索引 for ループ
| ... ;

列 for ループ = 'for', パターン束縛, 'in', ['reverse'], 式,
'do', 文 ;

集合 for ループ = 'for', 'all', パターン, 'in set', 式,
'do', 文 ;

索引 for ループ = 'for', 識別子, '=', 式, 'to', 式,
['by', 式], 'do', 文 ;

意味定義: *for* ループ文には 3 種類ある。索引を用いる for ループはほとんどの高水準プログラム言語において知られているものである。さらに、集合や列を用いる 2 つの for ループがある。これらは特に、集合 (または列) のすべての要素に対するアクセスが 1 つ 1 つ必要とされる場合に役立つ。

索引 *for* ループ文 は次の形式をもつ:

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

ここにおいて id は識別子、e1 と e2 はループの下限と上限を示す整数式、e3 はステップ幅を示す整数式、そして s は文でここで識別子 id を用いることができる。これは連続する文としての文 s の評価を表すが、現文脈は id の束縛により拡張されたものである。このように、最初に s が評価されるとき、 id は下限 e1 などの評価から戻ってきた値に束縛され、これは上限に達するまでつまり、 s > e2 となるまで同様に繰り返される。e1, e2 および e3 はループに入る前に評価されることに、注意したい。

集合 *for* ループ文 は次の形式をもつ:

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

ここで S は集合式である。文 s は、集合 S から e の束縛で拡張され部分列の値となった現環境において評価される。

列 *for* ループ文 は次の形式をもつ:

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

ここで l は列式である。文 s は、列 l から e の束縛と共に拡張され部分列値となった現環境で、評価される。キーワード reverse が用いられる場合は、列 l の要素は逆順にとられる。

例題: 操作 Remove は、数の列から与えられた 1 つの数のすべての出現を削除するための、 列 *for* ループの使用について説明している:

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

集合 *for* ループは、全グループの勝者集合を戻すように開発されることも可能である:

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

次の 索引 *for* ループの例は、古典的なバブルソートアルゴリズムである:

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

13.7 while ループ文

構文:

文 = ...
| while ループ
| ... ;

while ループ = 'while', 式, 'do', 文 ;

意味定義: *while* 文 に対する意味定義は、伝統的なプログラム言語からもってきた while 文 に対応する。*while* ループ の形式は次のとおり:

```
while e do
  s
```

ここで、 e はブール式、 s は文である。式 e が true と評価される限りは、本体文 s が評価される。

例題: *while* ループについては、相対誤差 e 内で実数 r の平方根を近似するニュートン法を用いた以下の例により、説明できる。

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

13.8 非決定文

構文:

文 = ...
| 非決定文
| ... ;

非決定文 = '||', '(', 文,
{ ',', 文 }, ')' ;

意味定義: 非決定文 は次の形式をもつ:

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

またこれは文構成要素 stmti の任意の (非決定) 順の実行を表すものである。しかしながら、文構成要素は同時に実行されるものではないことを書き留めておくべきであろう。インタープリタは、たとえその構成要素が非決定文と呼ばれたとしても、不十分決定の[21]意味定義を用いるということには注意しよう。

例題: 次の状態定義

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

を用いてバブルソートを行うために非決定文が使用できる:

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

ここで BubbleMin は、部分列 l(x,...,y) 内の最小値を "bubbles" した後に列の先頭にもってくる、また BubbleMax は、部分列 l(x,...,y) 内の最大値を "bubbles" した後に列の最終索引におく。BubbleMin は最小値の索引を見つけるために、最初に部分列を通して行う繰り返しにおいて働く。この索引の内容はその後、列 l(x) の先頭の内容と交換される。

[21] インタープリタの使用者がこれらの文要素が実行される順は知らないとしても、「プロジェクトオプション」の「乱数発生初期値」が用いられない限りは常に同じ順で実行される。

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

BubbleMax もまた同様に操作する。こちらは最大値の索引を見つけるために、部分列を通して反復を行い、その後この索引の内容を部分列の最後の要素の内容と交換する。

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

13.9 call 文

構文:

```
文 = ...
   | call 文
   | ... ;

call 文 = 名称, '(', [ 式リスト ], ')' ;
```

意味定義: *call* 文は次の形式をもつ:

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

call 文 は操作 opname を呼び、 その操作を評価した結果を返す。操作はグローバル変数を扱うことができるので、 *call* 文 は関数呼出しがそうするように必ずしも値を戻す必要はない。

例題: 以下の ResetStack 操作は変数を持たず、戻り値がない。一方、PopStack 操作はトップのスタック要素を返す。

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

この PopStack は以下のように定義することが出来る:

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

この stack はグローバル変数である。

13.10 return 文

構文:

```
文 = ...
   | return 文
   | ... ;

return 文 = 'return', [ 式 ] ;
```

意味定義: *return* 文 は操作内の式の値を戻す。値は与えられた文脈中で評価される。もし操作が値を返さないならば、式は省かれなければならない。*return* 文は次の形式をもつ:

```
return e
```

または

```
return
```

ここで式 e は操作の戻り値である。

例題: FunCall が関数呼出しであるの対し、以下に述べる例題 OpCall は操作呼出しである。*if* 文が 2 つに枝分かれした文だけを受け入れるため、 FunCall は *return* 文を用いることで文に “変換されて” いる。

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

13.11 例外処理文

構文:

```
文 = ...
   | always 文
   | trap 文
   | 再帰 trap 文
   | exit 文
   | ... ;
```

always 文 = 'always', 文, 'in', 文 ;

trap 文 = 'trap', パターン束縛, 'with', 文, 'in',
文 ;

再帰 trap 文 = 'tixe', trap 群, 'in', 文 ;

trap 群 = '{', パターン束縛, '|->', 文,
{ ',', パターン束縛, '|->', 文 }, '}' ;

exit 文 = 'exit', [式] ;

意味定義: 例外処理文は、仕様記述のなかで例外エラーを制御するために用いられる。このことは、仕様記述内で例外信号を送ることができなければならないことを意味する。これは *exit* 文を用いて行うことができ、次の形式をもつ:

```
exit e
```

または

```
exit
```

ここにおいて e はオプションの式である。式 e はどのような種類の例外が起きたのかを知らせるために用いられる。

always 文 は次の形式をもつ:

```
always s1 in
s2
```

ここで s1 と s2 は文である。最初に文 s2 が評価され、さらに例外が起きたかどうかにかかわらず文 s1 も評価される。*always* 文全体の結果値は、文 s1 の評価により決定される: もしここで例外を起こせばこの値が戻される、それ以外は文 s2 の評価の結果が返される。

trap 文 は、ある一定の条件が満たされたときに、処理文 s1 を評価する唯一のものである。これは次の形式をもつ:

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

ここで pat はある一定の例外を選択するために用いられるパターンまたは束縛であり、 s1 と s2 は文である。最初に文 s2 を評価するが、もし例外が発生しなければ s2 の評価結果が、 *trap* 文全体の結果値となる。例外が起きた場合には、 s2 の値はパターン pat と一致するか調べる。そこで一致するものがない場合、例外が *trap* 文全体の結果値として戻され、そうでない場合は文 s1 が評価され、この評価の結果がまた *trap* 文全体の結果値となる。

再帰 *trap* 文 は次の形式をもつ:

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

ここで pat1, ..., patn はパターンまたは束縛であり、s, s1, ..., sn は文である。最初に文 s が評価され、もし例外が起きなければ、その結果が完全な 再帰 *trap* 文の結果として戻される。そうでない場合、値はパターン pati の各々に順に一致させられる。一致するものが見つからなかった場合、 再帰 *trap* 文の結果として例外が戻される。一致するものが見つかれば、対応する文 si が評価される。これが例外を起こさない場合には、 si の評価の結果値は 再帰 *trap* 文の結果として戻される。それ以外の場合は、今度は新しい例外値 (si の評価の結果) をもとに、マッチングを再び始める。

例題: 多くのプログラムにおいて、1 つの操作にはメモリーを割り当てる必要がある。操作が完了した後には、このメモリーはそれ以上必要でなくなる。このことは *always* 文と共に実行される:

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

上記の例では、 *always* 文の本体文中で起きる可能性のある例外上では、何かを行うことはできない。*trap* 文 を用いることで、これらの例外を捉えることができる:

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

ここに *always* 文 中で起きるすべての例外は、 *trap* 文によって捉えられる。数個の例外値を区別したい場合には、ネストされた *trap* 文 かまたは再帰 *trap* 文を用いることができる:

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

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

操作 DoCommand では、メモリー割り当て中に *always* 文を用いるし、また操作 Example では、起きた例外はすべて 再帰 *trap* 文 によって捉えられる。値

<NOMEM> をもつ例外は、結果として -1 の戻り値となり例外は起きない。例外値が <BUSY> であるならば、再び操作 DoCommand の実行を試みる。これが例外を起こす場合には、 再帰 *trap* 文によってまた処理が行われる。他の例外はすべて、値 -2 を戻す結果となる。

13.12 error 文

構文:

文 = ...
| error 文
| ... ;

error 文 = 'error' ;

意味定義: *error* 文 は定義されていない式に相等する。文の結果は定義されないしこのことによりエラーが起きる、ということを明示的に述べるために用いられる。*error* 文 が評価されるときは、インタープリタは仕様記述の実行を停止し *error* 文 が評価されたことを報告する。

エラー文の実用的な使用となると、未定義式を用いた場合なので、事前条件の場合とは異なる: 事前条件の使用とは、操作が呼ばれたときは事前条件が満たされていると保証することは呼ぶ側の責任であることを意味する；error 文の使用では、エラー処理を取り扱うのは呼ばれた操作側の責任である。

例題: 100 ページ上の操作 SquareRoot は、平方する数は負かもしれないという可能性を排除するものでない。これについては操作 SquareRootErr で修正を行う:

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

```
    );
  return nextx
)
```

13.13 恒等文

構文: 文 = ...
| 恒等文 ;

恒等文 = 'skip' ;

意味定義: 恒等文 は何の評価も行われないという信号を送るために用いられる、

例題: 第 13.6 節の操作 Remove において、elem=z が明示的に述べられていない場合の for ループ内における操作の動き。以下の Remove2 がこれを示す。

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

ここで 恒等文を説明するために else 選択枝 を明示的に含めたが、ほとんどの場合 else 選択枝は含まれないため 恒等文 は暗黙に仮定されているのである。

13.14 仕様記述文

構文: 文 = ...
| 仕様記述文 ;

仕様記述文 = '[', 陰操作本体, ']' ;

意味定義: 仕様記述文は、事前条件や事後条件で表した文で期待効果を記述するために用いることができる。このように、操作定義に強要されることなく抽象的 (暗黙の) 仕様記述を許すことで、1 つの文の概略を捉える。仕様記述文は、暗黙に定義された陰操作 (第 12 節参照) の本体と同等である。したがって、仕様記述文を実行することはできない。

例題: 仕様記述文をバブルソートのバブル最大部を指定するために用いることができる:

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

(permutation は、もし 1 つの列が他方の並び替えであるならば真を返すような、2 つの列から結果をとる補助関数である。)

14 トップレベル仕様記述

前節で、型・式・文・関数、そして操作、という全ての VDM-SL の構成要素を述べてきた。これらのたくさんの構成要素は、トップレベルの VDM-SL 仕様記述の構成要素となることができる。トップレベル仕様記述は 2 つの方法により作り出される:

1. 仕様記述は、別々に指定される多くのモジュールに分けられるが、互いに依存することができる。
2. 仕様記述はフラットな方法で記述される。 すなわち、どのモジュールも使用されていない。

したがって、完全な仕様記述、または文書には、以下の構文がある。

構文:

```
文書 = 任意のモジュール, { 任意のモジュール }
     | 定義ブロック, { 定義ブロック } ;

任意のモジュール = モジュール
                 | 動的リンクモジュール ;
```

14.1 フラット仕様記述

前に述べたように、フラット仕様記述はモジュールを使用しない。これは、すべての構成要素が仕様記述中で使用することができることを意味する。フラットな場合では、文書は以下の構文を持っている:

```
文書 = ...
     | 定義ブロック, { 定義ブロック } ;

定義ブロック = 型定義群
             | 状態定義
             | 値定義群
             | 関数定義群
             | 操作定義群 ;
```

したがって、フラットな仕様記述はいくつかの定義ブロックで作られる。しかしながら、1 つの状態定義のみ許されている。以下はフラットトップレベル仕様記述に関する例である:

```
values

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

state St of
  l:seq1 of nat
end

functions

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

operations

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

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

14.2 構造仕様記述

標準の VDM-SL 言語への拡張として、モジュールを使用している VDM-SL 仕様記述を構造化することは可能である。この節で、トップレベル仕様記述を作成するためのモジュール使用方法を説明する。VDM-SL で提供されている構造化機能で以下が可能になる:

- モジュールから構成要素を輸出する
- モジュールから構成要素を輸入する
- 輸入時に構成要素名を変更する
- モジュールの状態を定義する

これらの普通のモジュールに加えて、いわゆる「ダイナミックリンクモジュール」(第 15 節参照) を使用することが可能である

14.2.1 モジュールのレイアウト

実際の機能を説明する前に、一般的なモジュールのレイアウトを説明する。モジュールは 3 つの要素から成る: モジュール宣言、インタフェース節、および定義節である。モジュール仕様記述の初期段階で定義部分を省略することは可能である。

モジュール宣言の中で、モジュールには名前が付けられる。その名前は完全な仕様記述の中では一意なモジュール名でなければならない。2 つ目 (インタフェース節) は、他のモジュールのモジュール関係を定義し、多くの節から成る。これらの節は以下の通りである:

- 輸入節 輸入節では、他のモジュールから使用されるすべての構成要素が述べられる。もしも、構成要素の名前を変更するなら、輸入節で行う。
- 輸出節 ここで、他のモジュールで使用されるすべての構成要素が定義される。1 つも輸出節が存在していないなら、そのモジュールは他のモジュールから使用することができない。

モジュール宣言の 3 つ目 (定義節) には全てのモジュール定義を含む。したがって、一般にはモジュールの構文は以下の通りである:

構文: モジュール = 'module', 識別子, インタフェース,
[モジュール本体], 'end', 識別子 ;

モジュール本体 = 'definitions', 定義ブロック, { 定義ブロック } ;

モジュールの使用を解説するために、フラットトップレベル仕様記述の例はやや手直しして記述する。フラット仕様記述の重要でない部分は分かりやすくするため省略する。

14.2.2 輸出節

構文: インタフェース = [輸入定義リスト],
輸出定義 ;

輸出定義 = 'exports', 輸出モジュールシグネチャ ;

輸出モジュールシグネチャ = 'all'
| 輸出シグネチャ,
{ 輸出シグネチャ } ;

輸出シグネチャ = 輸出型シグネチャ
| 値群シグネチャ
| 輸出関数シグネチャ
| 操作群シグネチャ ;

輸出型シグネチャ = 'types', 型輸出,
{ ';', 型輸出 }, [';'] ;

型輸出 = ['struct'], 名称 ;

値群シグネチャ = 'values', 値シグネチャ,
{ ';', 値シグネチャ }, [';'] ;

値シグネチャ = 名称リスト, ':', 型 ;

輸出関数シグネチャ = 'functions' 関数輸出,
{ ';', 関数輸出 }, [';'] ;

関数輸出 = 名称リスト, [型変数リスト], ':',
関数型 ;

関数群シグネチャ = 'functions' 関数シグネチャ,
{ ';', 関数シグネチャ }, [';'] ;

関数シグネチャ = 名称リスト, ':', 関数型 ;

操作群シグネチャ = 'operations' 操作シグネチャ,
{ ';', 操作シグネチャ }, [';'] ;

操作シグネチャ = 名称リスト, ':', 関数型 ;

意味定義: 輸出節は構成要素を他のモジュールから見えるよう公開するために使用する。モジュールで定義された構成要素のいくつか、またはすべてを輸出することが出来る。後者の場合では、キーワード all が使用される。しかしながら、輸入された構成要素はモジュールから輸出されない。構成要素の一部だけを輸出するなら、対応する適切なシグネチャの構造が公開される。

通常、構成要素が他のモジュールから見えるなら、その構造は、モジュール内で定義されていると見ることができる。しかしながら、型と操作にはいくつかの例外がある:

型: 型 T がモジュール A で定義され、この型がモジュール B で使用されるなら、モジュール A からの型は輸出しなければならないこれは、以下の 2 つの方法でできる:

1. その型の名前は輸出される
2. その型の構造は輸出される

型の名前だけを輸出するなら、もう一方のモジュールは型 T の値を作成することができない。これは、輸出しているモジュール (A) は T の表現に関する構成要素を用いて型 T の値を直接生成し操作するために関数と (または) 操作を与えなければならない。

私たちが struct というキーワードを使用することによって型の構造を輸出するなら、もう片方のモジュールは型 T の値 (もしもレコード型で

あるなら、この型に対して mk_キーワードと is_キーワードも使用できる) を生成し操作することができる。

型が不変条件も定義する場合、型の構造を輸出する場合にだけ、不変な述語関数を輸出する。

操作: モジュール内では、モジュールにとってグローバルな状態は定義することができる。モジュール内のすべての操作がその状態を操作することができる。モジュールから操作を輸出するなら、輸出しているモジュール (すなわち、それらが定義されるモジュールによる状態) で状態を操る。

輸出された関数、または操作が事前条件と (または) 事後条件を定義するなら、対応する述語関数 (第 6 節参照) も輸出する。

例: 銀行口座のモデルを考える。口座は所有者の名前、口座番号、銀行のキャッシュカードで残高、暗証番号を管理している口座の銀行支店によって特徴付けられている。以下のようにこれをモデル化する:

```
module BankAccount

  exports types digit; account
          functions digval: digit -> nat;
                    withdrawal: account * real -> account;
                    isPin: account * nat -> bool;
                    requestWithdrawal: account * nat -> bool
  definitions

  types

  digit = nat
  inv d == d < 10;

  account::  holder : seq1 of char
             number : seq1 of digit
             branchcode : seq1 of digit
             balance: real
             epin: nat
  inv mk_account(holder, number, branchcode,-,-) ==
```

```
    len number = 8 and len branchcode = 6

  functions

    digval : digit -> nat
    digval(d) == d;

    deposit: account * real -> account
    deposit(acc,r) ==
      mu(acc,balance |-> acc.balance + r);

    withdrawal : account * real -> account
    withdrawal (acc,r) ==
      mu(acc,balance |-> acc.balance - r);

    isPin : account * nat -> bool
    isPin(acc,ep) ==
      ep = acc.epin;

    requestWithdrawal : account * nat -> bool
    requestWithdrawal (acc,amt) ==
      acc.balance > amt

end BankAccount
```

このモジュール内で、2 つの型と 5 つの関数を輸出する。エンティティを列挙してから、輸出していることに注意する。しかし、struct というキーワードで使用されている digit や account は輸出されていない。つまり、account 値の内部は他のモジュールによってアクセスされないかもしれない。また、digit の不変条件もアクセスされないかもしれない。このようなアクセスが必要であるならば、その型は struct というキーワードで輸出されるべきであり、また、モジュール内の全ての構成要素は exports all 節を使用して輸出されるべきである。

以下に与えられたモジュール keypad は ATM のキーパッドインタフェースをモデル化する。その状態変数はユーザーによってキーパッドにタイプされたデータのバッファを維持する。

```
module Keypad

  imports
    from BankAccount types digit

  exports all

  definitions

  state buffer of
    data : seq of BankAccount`digit
  end

  operations

    DataAvailable : () ==> bool
    DataAvailable () ==
      return(data <> []);

    ReadData : () ==> seq of BankAccount`digit
    ReadData () ==
      return(data);

    WriteData : seq of BankAccount`digit ==> ()
    WriteData (d) ==
      data := data^d

end Keypad
```

このモジュールで、すべての構成要素を輸出する。定義された唯一のエンティティが、状態と操作であるので、これは、操作のすべてが輸入しているモジュールでアクセスされるかもしれないこと意味する。その状態は、輸入しているモジュールを利用しやすくないが、このモジュールは private のままである。しかしながら、状態のコンストラクタである mk_Keypad`buffer はアクセスできる。

14.2.3 輸入節

構文:

インタフェース = [輸入定義リスト],
輸出定義 ;

輸入定義リスト = 'Import', 輸入定義,
{ ',', 輸入定義 } ;

輸入定義 = 'from', 識別子, 輸入モジュールシグネチャ ;

輸入モジュールシグネチャ = 'all'
| 輸入シグネチャ,
{ 輸入シグネチャ } ;

輸入シグネチャ = 輸入型シグネチャ
| 輸入値シグネチャ
| 輸入関数シグネチャ
| 輸入操作シグネチャ ;

輸入型シグネチャ = 'types', 型輸入,
{ ';', 型輸入 }, [';'] ;

型輸入 = 名称, ['renamed', 名称]
| 型輸入, ['renamed', 名称] ;

輸入値シグネチャ = 'values', 値輸入,
{ ';', 値輸入 }, [';'] ;

値輸入 = 名称, [':', 型], ['renamed', 名称] ;

輸入関数シグネチャ = 'functions', 関数輸入,
{ ';', 関数輸入 }, [';'] ;

関数輸入 = 名称, [[型変数リスト],
':', 関数型], ['renamed', 名称] ;

輸入操作シグネチャ = 'operations', 操作輸入,
{ ';', 操作輸入 }, [';'] ;

操作輸入 = 名称, [':', 操作型],
['renamed', 名称] ;

意味定義: 輸入節はどの構成要素が目に見える構成要素のみ輸入することが出来るという制限を持った他のモジュールから使われるのかを述べるために使われる。モジュールからの全ての目に見える構成要素が利用されるならば、キーワードの all が使われる。但し、1 つ以上の構成要素が rename されない場合は除く。rename 時には、輸入された構成要素は輸出しているモジュール名だけ先行した元の名前の代わりに利用することができる新しい名前が与えられる。一般に、これは形式がある:

```
name renamed new_name
```

上記の name は輸入された構成要素の名前であり、また、new_name は構成要素の新しい名前である。このように、より意味のある名前が構成要素に与えられる。輸入しているモジュール内では、これ以上 DefModule'name(DefModule は定義モジュールの名前である) を言及することは不可能である。しかし、newname は可能である。

輸入節では、型情報を含むことが可能である。この情報は、完全なモジュールの静的意味定義チェックによってのみ使用される。型情報が与えられなかったら、静的意味定義も輸出しているモジュール内のこの情報を見つけることができる。(第 17 節参照)

型の輸入内で輸出しているモジュールから型定義が繰り返されるならば、struct キーワードで輸出している型が輸入される時には、この構成要素を使用するかもしれない。このような型の場合は、合成型であり、rename もされる。これは、さらにタグが rename される因果関係を持つ。

例: 私たちはカード番号と有効期限日から成り立つ銀行のキャッシュカードをモデル化することができる。 これは BankAccount というモジュールで定義された digit 型が必要である。また、それは同じモジュールから digval という関数を使用する。

```
module ATMCard

  imports
    from BankAccount types digit
```

```
                    functions digval renamed atmc_digval

  exports all

  definitions

  types

    digit = BankAccount`digit;

    atmc:: cardnumber : seq1 of digit
           expiry : digit * digit * digit * digit
    inv mk_atmc(cardnumber, mk_(m1,m2,-,-)) ==
        atmc_digval(m1) * 10 + atmc_digval(m2) <= 12 and
        len cardnumber >= 8

  functions

    getCardnumber : atmc -> seq1 of digit
    getCardnumber (atmc) ==
      atmc.cardnumber

end ATMCard
```

ここでは、atmc 型上の不変条件は、有効期限が有効日を表さなければならない。また、カードナンバーは少なくとも 8 桁以上でなければならないということである。degit が BankAccount モジュールから struct キーワードが輸出されていないので、ATMCard モジュール内の digit に対する不変条件にアクセスできない。しかしながら、これにもかかわらず、ATMCard 内で操作された digit 型のすべての値が不変条件を満たさなければならない。

15 動的リンクモジュール

動的リンクモジュールは VDM-SL で完全に仕様を定められたモジュールと C++ のコードとしてのみ利用可能なシステム全体の一部分との間のインタフェースを

記述するために使用された。仕様記述がインタープリタ、またはデバッグしている間、この機能は存在しているC++ライブラリをユーザが使用することを可能にする。この機能の使用法は[?]で詳細に説明する。動的リンクモジュールの一般的なレイアウトは普通のVDM-SLモジュールと似ている。それは3つの部分から成る:それは**モジュール宣言、インタフェース節、任意のライブラリ参照**である。

構文: 動的リンクモジュールのモジュール宣言は、単にモジュール名の後のキーワード、dlmoduleである。動的リンクモジュールのインタフェース節は普通のモジュールのインタフェース節より単純である。動的リンクモジュールに輸入することができる唯一の構成要素の種類は型である。このように輸入された型は、モジュールから輸出された値、関数や操作のシグネチャとして使用することが出来る。最終的にライブラリ参照 (‘uselib’ キーワードで特定される) は、このようなインタープリタが行われるライブラリからコードを利用する仕様記述の場合、インタープリタによって使われなければならない動的リンクC++ライブラリを特定するのに使われる。

動的リンクモジュールの構文は以下の通りである:

動的リンクモジュール = ‘dlmodule’, 識別子,
動的リンクインタフェース,
[use シグネチャ],
‘end’, 識別子 ;

動的リンクインタフェース = [動的リンク輸入定義リスト],
動的リンク輸出定義 ;

use シグネチャ = ‘uselib’, テキストリテラル ;

動的リンク輸入定義リスト = ‘imports’,
動的リンク輸入定義,
{ ‘,’, 動的リンク輸入定義 } ;

動的リンク輸入定義 = ‘from’, 識別子,
動的リンク輸入型シグネチャ群 ;

動的リンク輸入型シグネチャ群 = ‘types’, 名称,
{ ‘;’, 名称 }, [‘;’] ;

動的リンク輸出定義 = 'exports',
動的リンク輸出シグネチャ,
{ 動的リンク輸出シグネチャ } ;

動的リンク輸出シグネチャ = 値群シグネチャ
| 関数群シグネチャ
| 操作群シグネチャ ;

意味定義: インタフェース構成要素の意味定義は、普通のモジュールに対する意味定義と同じである。use シグネチャの意味定義は動的リンク C++ライブラリを作成するために使われた C++コンパイラによって与えられた。したがって、use シグネチャの中を参照する C++のコードは VDM-SL 水準において、直接意味定義を供給しない。

例: ここで紹介されている例は、[?] で利用されている。最初のモジュールは、MATH モジュールと CYLIO モジュールからの構成要素を輸入する。これら他のモジュールの両方がその後提示され、それらの両方が動的リンクモジュールである。

```
module CYLINDER
  imports
    from MATH
         functions
           ExtSin : real -> real
         values
           ExtPI : real,

    from CYLIO
         functions
           ExtGetCylinder : () -> CircCyl

         operations
           ExtShowCircCylVol : CircCyl * real ==> ()

  exports
    types
          CircCyl
```

```
  definitions
    types
        CircCyl :: rad    : real
                   height : real
                   slope  : real

    functions
        CircCylVol : CircCyl -> real
        CircCylVol(cyl) ==
          MATH`ExtPI * cyl.rad * cyl.rad * cyl.height *
          MATH`ExtSin(cyl.slope)

    operations
        CircCyl : () ==> ()
        CircCyl() == ( let cyl = CYLIO`ExtGetCylinder() in
                         let vol = CircCylVol(cyl) in
                           CYLIO`ExtShowCircCylVol(cyl, vol))
end CYLINDER
```

The MATH モジュールは次のように定義される:

```
dlmodule MATH
  exports
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

    values
      ExtPI : real

    uselib
      "libmath.so"

end MATH
```

The CYLIO module is defined as:

```
dlmodule CYLIO
  imports
    from CYLINDER
      types
        CircCyl

  exports
    functions
      ExtGetCylinder : () -> CircCyl

    operations
      ExtShowCircCylVol : CircCyl * real ==> ()
  88

  uselib
     "libcylio.so"

end CYLIO
```

VDM-SL インタープリタでこのようなモジュールを使用する方法は [?] 中に述べられる。

16 VDM-SL と ISO /VDM-SL の相違点

VDM-SL のこの版は ISO/VDM-SL 標準を基本とするが、少しだけ異なる点がある。これらの相違は構文上および意味定義上の両面にあり、主に言語の拡張に起因するもので VDM-SL 構成要素を実行可能なものにしようとする要求に原因がある[22]。

VDM-SL と ISO/VDM-SL の主要な違いは VDM-SL の構造化の利用である。これは構文の違いが原因である。

VDM-SL のフラット部分に対して、ISO/VDM-SL は以下の違いが存在している:

[22] ここでの意味定義とはインタープリタの意味定義のことである。

構文上の相違:

- 標準では部分列構成要素間 (たとえば関数定義間) の分離符としてセミコロン (";") が使われる。VDM-SL ではこの規則に、このような構成要素の列の最後尾に随意でセミコロンをつけることができることを付け加える。この変更は以下に述べる構文上の定義に対しても適用される (付録 A を参照) : **状態定義**, **型定義**, **値定義**, **関数定義**, **操作定義**, *def***式**, *def***文**, **ブロック文**。
- 陽関数定義および操作定義においては、 VDM-SL におけるオプションの事後条件を指定することが可能である (第 6 節と第 12 節、または第 A.3.4 節か A.3.5 節を参照)。
- 陽関数定義および操作定義の本体は 節 is not yet specified を用いて仮の方法で指定することができる。
- 陽関数定義および操作定義に対する拡張形式が組み入れられてきた。拡張は、陰仕様定義に用いられたのと同様の見出しの使用を、関数と操作の定義で可能にするものである。これはまず最初に陰仕様定義を書くことをより容易にし、そしてアルゴリズム部分を後に続けて加えることを容易にする。加えて、結果識別子の型ペアは 2 つ以上の識別子とともに働くために生成されたものである。

  フラット仕様記述では、definitions というキーワードは使用されない。このように、いくつかのファイルの上にフラット仕様記述を分配することができる。しかしながら、モジュール内では、定義部は definitions というキーワードで始まらなければならない。(第 14.1 節参照)

  VDM-SL は *specification statement* (仕様記述文) で拡張された。
- *if* 文 の中で "else" 部はオプションである (第 13.5 節または第 A.6.3 節参照)
- 空集合と空列は直接にパターンとして用いることができる (第 8 節か A.7.1 節参照)
- "制限する写像定義域" と "制限される写像定義域" は適切にグループ化される (第 C.7 節参照)
- 写像型構成子に対する演算子優先順位は標準と異なる (第 C.8 章参照)
- VDM++ においては、組選択、型判断、事前条件式が加わる
- VDM++ においては、原子代入文が加わる

意味定義上の相違 (wrt. インタープリタ):

- VDM-SL は条件論理によってのみ動作する (第 4.1.1 節参照)。
- グローバル状態の初期化を、特別に構造的な方法で書かなければならない。少なくとも、モジュールからある操作が行われたら、モジュールの状態は初期化のみされることに注意する。(第 11 節参照)。
- VDM-SL においては、相互に再帰する値定義は実行不可能であり、またそれらは用いられる前に定義されているように順序付けられなければならない (第 10 節参照)。
- *let* 文と *let* 式におけるローカル定義は再帰的に定義されることは許されない。さらにそれらは用いられる前に定義されているように順序付けられなければならない (第 7.1 節と第 13.1 節参照)。
- VDM-SL における数値型 rat は型 real と同じ型を表す (第 4.1.2 節参照)。
- ISO/VDM-SL において用いられるインタープリタ実行の自由度の 2 つの形式 は '劣決定系' と '非決定系である。ISO/VDM-SL の操作における自由度では、関数に対して劣決定系である場合はすべて非決定系となる。VDM-SL においては、操作と関数の両方における自由度は劣決定系である。これはしかしながら、標準と一致するものでもある、なぜならインタープリタは単に仕様記述に対する可能な様式のひとつに相応するからである。

17 静的意味

構文規則に従った構文的に正しい VDM 仕様記述でも、必ずしも言語の型やスコープの規則に従ってはいない。VDM 仕様記述が良形であるかについては、**静的意味チェッカー**によって検査することができる。Toolbox にはこのような静的意味チェッカー (プログラム言語では通常は型検査と呼ばれる) が存在する。

一般的に、与えられた VDM 仕様記述が良形か否かは、静的に決定可能であるとはいえない。VDM-SL の静的意味は他の言語の静的意味とは異なる。他の言語は絶対的に良形だとはいえない仕様記述のみを拒絶し、また絶対的に良形である仕様記述のみを受け入れる。VDM-SL の静的意味は、VDM 仕様記述に対する**良形である度合**があるものと考える。このような良形である度合は、仕様記述

が絶対的に良形であるか、絶対的に良形でないか、恐らく良形か、を示すものである。

このことは Toolbox において、静的意味チェッカーが、恐らくの正しさかあるいは絶対的な正しさかどちらかをチェックできることを意味する。しかしながら、ただのほんとに単純な仕様記述のみが、絶対的良形であることのチェックにパスすることができるのだ、ということを記しておくべきであろう。したがって、実際的な使用においては “恐らく良形” がもっとも役に立つ。

恐らく良形のチェックと絶対的良形のチェックの間の相違は、以下に続く VDM 仕様記述の断片によって実例をもって示すことができる:

```
if a = true
then a + 1
else not a
```

ここで a は型 nat | bool (nat と bool の合併型) をもっている。もし a が true と等しいならば、そのとき a に 1 を加えることは不可能となるであろうから、この式は悪形であると読者は簡単に理解することができる。しかしながら、このような式は恣意的に複雑になり得るので、一般的にはこれが静的にチェックされる可能性はない。この特別な例において、絶対的な良形であることは false となる一方で、恐らく良形であることは true となるであろう。

参考文献

[1] Bicarregui, J., Fitzgerald, J., Lindsay, P., Moore, R., and Ritchie, B. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[2] Dawes, J. *The VDM-SL Reference Guide*. Pitman, 1991. ISBN 0-273-03151-1.

[3] Fitzgerald, J., and Jones, C. *Proof in VDM: case studies*. Springer-Verlag FACIT Series, 1998, ch. Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant.

[4] Jones, C. B. *Systematic Software Development Using VDM*, second ed. Prentice-Hall International, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-880733-7.

[5] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[6] Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1991.

A VDM-SL 構文

この付録は VDM-SL の完全な構文を明確にするものである。

A.1 文書

文書 = 任意のモジュール, { 任意のモジュール }
| 定義ブロック, { 定義ブロック } ;

任意のモジュール = モジュール
| 動的リンクモジュール ;

A.2 モジュール

この全体の小区分は VDM-SL 標準の最新版で存在しない。

標準ではない

モジュール = 'module', 識別子, インタフェース,
[モジュール本体], 'end', 識別子 ;

インタフェース = [輸入定義リスト],
輸出定義 ;

輸入定義リスト = 'imports', 輸入定義,
{ ',', 輸入定義 } ;

輸入定義 = 'from', 識別子, 輸入モジュールシグネチャ ;

輸入モジュールシグネチャ = 'all'
| 輸入シグネチャ, { 輸入シグネチャ } ;

```
輸入シグネチャ = 輸入型シグネチャ
               | 輸入値シグネチャ
               | 輸入関数シグネチャ
               | 輸入操作シグネチャ ;

輸入型シグネチャ = 'types', 型輸入,
                   { ';', 型輸入 }, [ ';' ] ;

型輸入 = 名称, [ 'renamed', 名称 ]
       | 型定義, [ 'renamed', 名称 ] ;

輸入値シグネチャ = 'values', 値輸入,
                   { ';', 値輸入 }, [ ';' ] ;

値輸入 = 名称, [ ':', 型 ], [ 'renamed', 名称 ] ;

輸入関数シグネチャ = 'functions', 関数輸入,
                     { ';', 関数輸入 }, [ ';' ] ;

関数輸入 = 名称, [ [ 型変数リスト ], ':', 関数型 ],
           [ 'renamed', 名称 ] ;

輸入操作シグネチャ = 'operations', 操作輸入,
                     { ';', 操作輸入 }, [ ';' ] ;

操作輸入 = 名称, [ ':', 操作型 ], [ 'renamed', 名称 ] ;

輸出定義 = 'exports', 輸出モジュールシグネチャ ;

輸出モジュールシグネチャ = 'all'
                         | 輸出シグネチャ,
                           { 輸出シグネチャ } ;
```

```
輸出シグネチャ = 輸出型シグネチャ
               | 値群シグネチャ
               | 輸出関数シグネチャ
               | 操作群シグネチャ ;

輸出型シグネチャ = 'types', 型輸出,
                   { ';', 型輸出 }, [ ';' ] ;

型輸出 = [ 'struct' ], 名称 ;

値群シグネチャ = 'values', 値シグネチャ,
                 { ';', 値シグネチャ }, [ ';' ] ;

値シグネチャ = 名称リスト, ':', 型 ;

輸出関数シグネチャ = 'functions' 関数輸出,
                     { ';', 関数輸出 }, [ ';' ] ;

関数輸出 = 名称リスト, [ 型変数リスト ], ':',
           関数型 ;

関数群シグネチャ = 'functions' 関数シグネチャ,
                   { ';', 関数シグネチャ }, [ ';' ] ;

関数シグネチャ = 名称リスト, ':', 関数型 ;

操作群シグネチャ = 'operations' 操作シグネチャ,
                   { ';', 操作シグネチャ }, [ ';' ] ;

操作シグネチャ = 名称リスト, ':', 操作型 ;

動的リンクモジュール = 'dlmodule', 識別子,
                       動的リンクインタフェース,
                       [ use シグネチャ ],
                       'end', 識別子 ;
```

動的リンクインタフェース = [動的リンク輸入定義リスト],
動的リンク輸出定義 ;

use シグネチャ = 'uselib', テキストリテラル ;

動的リンク輸入定義リスト = 'imports',
動的リンク輸入定義,
{ ',', 動的リンク輸入定義 } ;

動的リンク輸入定義 = 'from', 識別子,
動的リンク輸入型シグネチャ群 ;

動的リンク輸入型シグネチャ群 = 'types', 名称,
{ ';', 名称 }, [';'] ;

動的リンク輸出定義 = 'exports',
動的リンク輸出シグネチャ,
{ 動的リンク輸出シグネチャ } ;

動的リンク輸出シグネチャ = 値群シグネチャ
| 関数群シグネチャ
| 操作群シグネチャ ;

A.3 定義

モジュール本体 = 'definitions', 定義ブロック, { 定義ブロック } ;

定義ブロック = 型定義群
| 状態定義群
| 値定義群
| 関数定義群
| 操作定義群 ;

A.3.1 型定義

型定義群 = 'types', [型定義, { ';', 型定義 }, [';']] ; Non standard

型定義 = 識別子, '=', 型, [不変条件]
| 識別子, '::', 項目リスト, [不変条件] ;

型 = 括弧型
| 基本型
| 引用型
| レコード型
| 合併型
| 組型
| 選択型
| 集合型
| 列型
| 写像型
| 部分関数型
| 型名称
| 型変数 ;

括弧型 = '(', 型, ')' ;

基本型 = 'bool' | 'nat' | 'nat1' | 'int' | 'rat'
| 'real' | 'char' | 'token' ;

引用型 = '<', 引用リテラル, '>' ;

レコード型 = 'compose', 識別子, 'of', 項目リスト, 'end' ;

項目リスト = { 項目 } ;

項目 = [識別子, ‘:’], 型
| [識別子, ‘:-’], 型 ;

合併型 = 型, ‘|’, 型, { ‘|’, 型 } ;

組型 = 型, ‘*’, 型, { ‘*’, 型 } ;

選択型 = ‘[’, 型, ‘]’ ;

集合型 = ‘set of’, 型 ;

列型 = 空列を含む列型
| 空列を含まない列型 ;

空列を含む列型 = ‘seq of’, 型 ;

空列を含まない列型 = ‘seq1 of’, 型 ;

写像型 = 一般写像型
| 1 対 1 写像型 ;

一般写像型 = ‘map’, 型, ‘to’, 型 ;

1 対 1 写像型 = ‘inmap’, 型, ‘to’, 型 ;

関数型 = 部分関数型
| 全関数型 ;

部分関数型 = 任意の型, ‘->’, 型 ;

全関数型 = 任意の型, ‘+>’, 型 ;

任意の型 = 型
| '(', ')' ;

型名称 = 名称 ;

型変数 = 型変数識別子 ;

A.3.2 状態定義

状態定義 = 'state', 識別子, 'of', 項目リスト,
[不変条件], [初期化], 'end', [';'] ; Non standard

不変条件 = 'inv', 不変条件初期関数 ;

初期化 = 'init', 不変条件初期関数 ;

不変条件初期関数 = パターン, '==', 式 ;

A.3.3 値定義

値定義群 = 'values', [値定義,
{ ';', 値定義 }, [';']] ; Non standard

値定義 = パターン, [':', 型], '=', 式 ; Non standard

A.3.4 関数定義

関数定義群 = 'functions', [関数定義,
{ ';', 関数定義 }, [';']] ; Non standard

関数定義 = 陽関数定義
| 陰関数定義
| 拡張陽関数定義 ; Non standard

陽関数定義 = 識別子, [型変数リスト], ':',
関数型,
識別子, パラメーターリスト,
'==', 関数本体,
['pre', 式],
['post', 式],
['measure', 名称] ; Non standard

陰関数定義 = 識別子, [型変数リスト],
パラメーター型,
識別子型ペアリスト,
['pre', 式],
'post', 式 ;

拡張陽関数定義 = 識別子, [型変数リスト], Non standard
パラメーター型,
識別子型ペアリスト,
'==', 関数本体,
['pre', 式],
['post', 式] ;

型変数リスト = '[', 型変数識別子,
{ ',', 型変数識別子 }, ']' ;

識別子型ペア = 識別子, ‘:’, 型 ;

パラメーター型 = ‘(’, [パターン型ペアリスト], ‘)’ ;

識別子型ペアリスト = 識別子, ‘:’, 型,
{ ‘,’, 識別子, ‘:’, 型 } ;

パターン型ペアリスト = パターンリスト, ‘:’, 型,
{ ‘,’, パターンリスト,‘:’, 型 } ;

パラメーターリスト = パラメーター群, { パラメーター群 } ;

パラメーター群 = ‘(’, [パターンリスト], ‘)’ ;

関数本体 = 式
| ‘is not yet specified’ ; Non standard

A.3.5 操作定義

操作定義群 = ‘operations’, [操作定義,
{ ‘;’, 操作定義 }, [‘;’]] ; Non standard

操作定義 = 陽操作定義
| 陰操作定義
| 拡張陽操作定義 ; Non standard

陽操作定義 = 識別子, ‘:’, 操作型,
識別子, パラメーター群,
‘==’, 操作本体,
[‘pre’, 式],
[‘post’, 式] ;

Non standard

陰操作定義 = 識別子, パラメーター型,
[識別子型ペアリスト],
陰操作本体 ;

陰操作本体 = [外部節],
['pre', 式],
'post', 式,
[例外] ;

Non standard

拡張陽操作定義 = 識別子, パラメーター型,
[識別子型ペアリスト],
'==', 操作本体,
[外部節],
['pre', 式],
['post', 式],
[例外] ;

操作型 = 任意の型, '==>', 任意の型 ;

操作本体 = 文
| 'is not yet specified' ;

Non standard

外部節 = 'ext', var 情報, { var 情報 } ;

var 情報 = モード, 名称リスト, [':', 型] ;

モード = 'rd' | 'wr' ;

例外 = 'errs', エラーリスト ;

エラーリスト = エラー, { エラー } ;

エラー = 識別子, ':', 式, '->', 式 ;

A.4 式

式リスト = 式, { ',', 式 } ;

式 = 括弧式
| let 式
| let be 式
| def 式
| if 式
| cases 式
| 単項式
| 2 項式
| 限量式
| iota 式
| 集合列挙
| 集合内包
| 集合範囲式
| 列列挙
| 列内包
| 部分列
| 写像列挙
| 写像内包
| 組構成子
| レコード構成子
| レコード修正子
| 適用
| 項目選択
| 組選択
| 関数型インスタンス化
| ラムダ式
| 一般 is 式
| 未定義式
| 事前条件式 Non standard
| 名称
| 旧名称
| 記号リテラル ;

A.4.1 括弧式

括弧式 = '(', 式, ')' ;

A.4.2 ローカル束縛式

let 式 = 'let', ローカル定義, { ',', ローカル定義 },
'in', 式 ;

let be 式 = 'let', 束縛, ['be', 'st', 式], 'in',
式 ;

def 式 = 'def', パターン束縛, '=', 式,
{ ';', パターン束縛, '=', 式 }, [';'],
'in', 式 ;

Non standard

A.4.3 条件式

if 式 = 'if', 式, 'then', 式,
{ elseif 式 },
'else', 式 ;

elseif 式 = 'elseif', 式, 'then', 式 ;

cases 式 = 'cases', 式, ':',
cases 式選択肢群,
[',', others 式], 'end' ;

cases 式選択肢群 = cases 式選択肢,
{ ',', cases 式選択肢 } ;

cases 式選択肢 = パターンリスト, '->', 式 ;

others 式 = 'others', '->', 式 ;

A.4.4 単項式

単項式 = 接頭辞式
| 逆写像 ;

接頭辞式 = 単項演算子, 式 ;

単項演算子 = 正符号
| 負符号
| 算術絶対値
| 底値
| 否定
| 集合濃度
| 有限べき集合
| 分配的集合合併
| 分配的集合共通部分
| 列先頭
| 列尾部
| 列長
| 列要素
| 列索引
| 分配的列連結
| 写像定義域
| 写像値域
| 分配的写像併合 ;

正符号 = '+' ;

負符号 = '-' ;

算術絶対値 = 'abs' ;

底値 = 'floor' ;

否定 = 'not' ;

集合濃度 = ‘card’ ;

有限べき集合 = ‘power’ ;

分配的集合合併 = ‘dunion’ ;

分配的集合共通部分 = ‘dinter’ ;

列先頭 = ‘hd’ ;

列尾部 = ‘tl’ ;

列長 = ‘len’ ;

列要素 = ‘elems’ ;

列索引 = ‘inds’ ;

分配的列連結 = ‘conc’ ;

写像定義域 = ‘dom’ ;

写像値域 = ‘rng’ ;

分配的写像併合 = ‘merge’ ;

逆写像 = ‘inverse’, 式 ;

A.4.5 2 項式

2 項式 = 式, 2 項演算子, 式 ;

2 項演算子 = 加算
| 減算
| 乗算
| 除算
| 整数除算
| 剰余算
| 法算
| より小さい
| より小さいか等しい
| より大きい
| より大きいか等しい
| 相等
| 不等
| 論理積
| 論理和
| 含意
| 同値
| 帰属
| 非帰属
| 包含
| 真包含
| 集合合併
| 集合差
| 集合共通部分
| 列連結
| 写像修正または列修正
| 写像併合
| 写像定義域限定
| 写像定義域削減
| 写像値域限定
| 写像値域削減
| 合成
| 反復 ;

加算 = '+' ;

減算 = '-' ;

乗算 = '*' ;

除算 = '/' ;

整数除算 = 'div' ;

剰余算 = 'rem' ;

法算 = 'mod' ;

より小さい = '<' ;

より小さいか等しい = '<=' ;

より大きい = '>' ;

より大きいか等しい = '>=' ;

相等 = '=' ;

不等 = '<>' ;

論理和 = 'or' ;

論理積 = 'and' ;

含意 = '=>' ;

同値 = '<=>' ;

帰属 = 'in set' ;

非帰属 = 'not in set' ;

包含 = 'subset' ;

真包含 = 'psubset' ;

集合合併 = 'union' ;

集合差 = '\' ;

集合共通部分 = 'inter' ;

列連結 = '^' ;

写像修正または列修正 = '++' ;

写像併合 = 'munion' ;

写像定義域限定 = '<:' ;

写像定義域削減 = '<-:' ;

写像値域限定 = ':>' ;

写像値域削減 = ':->' ;

合成 = 'comp' ;

反復 = '**' ;

A.4.6 限量式

限量式 = 全称限量式
| 存在限量式
| 1 存在限量式 ;

全称限量式 = 'forall', 束縛リスト, '&', 式 ;

存在限量式 = 'exists', 束縛リスト, '&', 式 ;

1 存在限量式 = 'exists1', 束縛, '&', 式 ;

A.4.7 iota 式

iota 式 = 'iota', 束縛, '&', 式 ;

A.4.8 集合式

集合列挙 = '{', [式リスト], '}' ;

集合内包 = '{', 式, 'l', 束縛リスト,
['&', 式], '}' ;

集合範囲式 = '{', 式, ',', '...', ',',
式, '}' ;

A.4.9 列式

列列挙 = '[', [式リスト], ']' ;

列内包 = '[', 式, 'l', 集合束縛,
['&', 式], ']' ;

部分列 = 式, '(', 式, ',', '...', ',',
式, ')' ;

A.4.10 写像式

写像列挙 = '{', 写, { ',', 写 }, '}'
| '{', '|->', '}' ;

写 = 式, '|->', 式 ;

写像内包 = '{', 写, '|', 束縛リスト,
['&', 式], '}' ;

A.4.11 組構成子式

組構成子 = 'mk_', '(', 式, ',', 式リスト, ')' ;

A.4.12 レコード式

レコード構成子 = 'mk_',[23] 名称, '(', [式リスト], ')' ;

レコード修正子 = 'mu', '(', 式, ',',
レコード修正,
{ ',', レコード修正 }, ')' ;

レコード修正 = 識別子, '|->', 式 ;

A.4.13 適用式

適用 = 式, '(', [式リスト], ')' ;

項目選択 = 式, '.', 識別子 ;

組選択 = 式, '.#', 数字 ;

関数型インスタンス化 = 名称, '[', 型, { ',', 型 }, ']' ;

[23] **Note:** 境界文字は許されない

A.4.14 ラムダ式

ラムダ式 = 'lambda', 型束縛リスト, '&', 式 ;

A.4.15 narrow 式

narrow 式 = 'narrow_', '(', 式, ',', 型, ')' ;

A.4.16 is 式

一般 is 式 = is 式
| 型判定 ;

is 式 = 'is_',[24] 名称, '(', 式, ')'
| is 基本型, '(', 式, ')' ;

型判定 = 'is_', '(', 式, ',', 型, ')' ;

A.4.17 未定義式

未定義式 = 'undefined' ; Non standard

A.4.18 事前条件式

事前条件式 = 'pre_', '(', 式,
[{ ',', 式 }], ')' ;

[24] **Note:** 境界文字は許されない

A.4.19 名称

名称 = 識別子, [‘‘’, 識別子] ;

名称リスト = 名称, { ‘,’, 名称 } ;

旧名称 = 識別子, ‘~’ ;

A.5 状態指示子

状態指示子 = 名称
| 項目参照
| 写像参照または列参照 ;

項目参照 = 状態指示子, ‘.’, 識別子 ;

写像参照または列参照 = 状態指示子, ‘(’, 式, ‘)’ ;

A.6 文

文 = let 文
| let be 文
| def 文
| ブロック文
| 代入文
| if 文
| cases 文
| 列 for ループ
| 集合 for ループ
| 索引 for ループ
| while ループ
| 非決定文
| call 文

| 仕様記述文
| return 文
| always 文
| trap 文
| 再帰 trap 文
| exit 文
| error 文
| 恒等文 ;

Non standard

A.6.1 ローカル束縛文

let 文 = 'let', ローカル定義, { ',', ローカル定義 },
'in', 文 ;

ローカル定義 = 値定義
| 関数定義 ;

let be 文 = 'let', 束縛, ['be', 'st', 式], 'in',
文 ;

def 文 = 'def', 相等定義,
{ ';', 相等定義 }, [';'],
'in', 文 ;

Non standard

相等定義 = パターン束縛, '=', 式 ;

A.6.2 ブロック文と代入文

ブロック文 = '(', { dcl 文 },
文, { ';', 文 }, [';'], ')' ;

Non standard

dcl 文 = 'dcl', 代入定義,
{ ',', 代入定義 }, ';' ;

代入定義 = 識別子, ':', 型, [':=', 式] ;

一般代入文 = 代入文
| 多重代入文 ;

代入文 = 状態指示子, ':=', 式 ;

多重代入文 = 'atomic', '(' 代入文, ';',
代入文,
{ ';', 代入文 }')' ;

A.6.3 条件文

if 文 = 'if', 式, 'then', 文,
{ elseif 文 },
['else', 文] ;

elseif 文 = 'elseif', 式, 'then', 文 ;

cases 文 = 'cases', 式, ':',
cases 文選択肢群,
[',', others 文], 'end' ;

cases 文選択肢群 = cases 文選択肢,
{ ',', cases 文選択肢 } ;

cases 文選択肢 = パターンリスト, '->', 文 ;

others 文 = 'others', '->', 文 ;

A.6.4 ループ文

列 for ループ = 'for', パターン束縛, 'in', ['reverse'],
式, 'do', 文 ;

集合 for ループ = 'for', 'all', パターン, 'in set', 式,
'do', 文 ;

索引 for ループ = 'for', 識別子, '=', 式, 'to', 式,
['by', 式],
'do', 式 ;

while ループ = 'while', 式, 'do', 式 ;

A.6.5 非決定文

非決定文 = '||', '(', 文,
{ ',', 文 }, ')' ;

A.6.6 call 文と return 文

call 文 = 名称, '(',
[式リスト], ')' ;

return 文 = 'return', [式] ;

A.6.7 仕様記述文

仕様記述文 = '[', 陰操作本体, ']' ; Non standard

A.6.8 例外処理文

always 文 = 'always', 文, 'in', 文 ;

trap 文 = 'trap', パターン束縛, 'with', 文,
'in', 文 ;

再帰 trap 文 = 'tixe', trap 群, 'in', 文 ;

trap 群 = '{', パターン束縛, '|->', 文,
{ ',', パターン束縛, '|->', 文 }, '}' ;

exit 文 = 'exit', [式] ;

A.6.9 error 文

error 文 = 'error' ; Non standard

A.6.10 恒等文

恒等文 = 'skip' ;

A.7 パターンと束縛

A.7.1 パターン

パターン = パターン識別子
| 一致値
| 集合列挙パターン
| 集合合併パターン
| 列列挙パターン

| 列連結パターン
| 写像列挙パターン
| 写像併合パターン
| 組パターン
| レコードパターン ;

パターン識別子 = 識別子 | '-' ;

一致値 = '(', 式, ')'
| 記号リテラル ;

集合列挙パターン = '{', [パターンリスト], '}' ; Non standard

集合合併パターン = パターン, 'union', パターン ;

列列挙パターン = '[', [パターンリスト], ']' ; Non standard

列連結パターン = パターン, '^', パターン ;

写像列挙パターン = '{', [写パターンリスト], '}' ;

写パターンリスト = 写パターン, { ',', 写パターン } ;

写パターン = パターン, '|->', パターン ;

写像併合パターン = パターン, 'munion', パターン ;

組パターン = 'mk_', '(', パターン, ',', パターンリスト, ')' ;

レコードパターン = 'mk_',[25] 名称, '(', [パターンリスト], ')' ;

パターンリスト = パターン, { ',', パターン } ;

[25] **Note:** 境界文字は許されない

A.7.2 束縛

パターン束縛 = パターン | 束縛 ;

束縛 = 集合束縛 | 型束縛 ;

集合束縛 = パターン, 'in set', 式 ;

型束縛 = パターン, ':', 型 ;

束縛リスト = 多重束縛, { ',', 多重束縛 } ;

多重束縛 = 多重集合束縛
| 多重型束縛 ;

多重集合束縛 = パターンリスト, 'in set', 式 ;

多重型束縛 = パターンリスト, ':', 型 ;

型束縛リスト = 型束縛, { ',', 型束縛 } ;

B 語彙

B.1 文字

文字集合は、この文書で用いられた文字形式と共に、11 表に示される。VDM-SL 標準において文字は次のとおりに定義される:

文字 = 通常文字
| キーワード文字
| 識別文字

| ギリシャ文字
| アラビア数字
| 境界文字
| その他の文字
| 分離符 ;

通常文字とキーワード文字は 11 表 に表示されている (この文書ではキーワード文字は単に相当するアルファベット小文字を用いている)。識別文字は相当するアルファベットの大文字と小文字を用いるが引用リテラルは “<” で始め “>” で閉じる (引用文字にはアンダーバーや数字も用いることができることに注目)。ギリシャ文字も数値記号 “#” の後に相当する文字を続けることで使用できる (この情報は LaTeX pretty printer で用いられ、ギリシャ文字が提示できるようになっている)。全境界文字 (標準 ASCII 版のもの) は表 11 に記述されている。標準では境界文字とそれらの組み合わせには違いをもたせている。ここではこれを違いとして扱わない。数学構文中のいくつかの境界文字が、ここで用いられている ASCII 構文中ではキーワードとなることにも気づいてもらいたい。

文字:

```
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
```

漢字 ハングル文字

キーワード文字:

```
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
```

境界文字:

```
,    :    ;    =    (    )    |    -    [    ]
{    }    +    /    <    >    <=   >=   <>   .
*    ->   +>   ==>  ||   =>   <=>  |->  <:   :>
<-:  :->  &    ==   **   ^    ++
```

アラビア数字:

```
0  1  2  3  4  5  6  7  8  9
```

16 進数字:

```
0  1  2  3  4  5  6  7  8  9
A  B  C  D  E  F
a  b  c  d  e  f
```

8 進数字:

```
0  1  2  3  4  5  6  7
```

その他の文字:

```
_  `  '  "  @  ~
```

改行:

空白:

これらはグラフィックな形態をもたないが、空白と行替えを組み合わせる。ここに 2 つの分離符がある: 行替えしない (空白) と行替えを行う (改行) である。

表 11: 文字集合

B.2 記号

以下のような種類の記号が存在する: キーワード、境界文字、記号リテラル、そしてコメント。文字から記号への変換も以下の規則のもとに与えられている; これらは構文定義と同じ表記を用いるが、意味においては続く終了符との間に分離符がないという点で異なる。それ例外の曖昧さが生ずる場所では、2 つの連続する記号は分離符によって分離されていなければならない。

```
キーワード = 'abs' | 'all' | 'always' | 'and' | 'as' | 'atomic' | 'be' | 'bool' | 'by'
           | 'card' | 'cases' | 'char' | 'comp' | 'compose' | 'conc'
           | 'dcl' | 'def' | 'definitions' | 'dinter' | 'div' | 'dlmodule'
           | 'do' | 'dom' | 'dunion' | 'elems' | 'else' | 'elseif' | 'end'
           | 'error' | 'errs' | 'exists' | 'exists1' | 'exit' | 'exports' | 'ext'
           | 'false' | 'floor' | 'for' | 'forall' | 'from' | 'functions'
           | 'hd' | 'if' | 'imports' | 'in' | 'inds' | 'init' | 'inmap'
           | 'int' | 'inter' | 'inv' | 'inverse' | 'iota' | 'is_'
           | 'lambda' | 'len' | 'let' | 'make_' | 'map' | 'narrow_' | 'measure'
           | 'merge' | 'mod' | 'module'
           | 'mu_' | 'munion' | 'nat' | 'nat1' | 'nil' | 'not' | 'of'
           | 'operations' | 'or' | 'others' | 'post'
           | 'power' | 'pre' | 'psubset' | 'rat' | 'rd' | 'real' | 'rem'
           | 'renamed' | 'return' | 'reverse' | 'rng' | 'seq' | 'seq1'
           | 'set' | 'skip' | 'specified' | 'st' | 'state' | 'struct'
           | 'subset' | 'then' | 'tixe' | 'tl' | 'to' | 'token' | 'trap'
           | 'true' | 'types' | 'undefined' | 'union' | 'uselib'
           | 'values' | 'while' | 'with' | 'wr' | 'yet'
           | 'RESULT' ;

分離符 = 改行 | 空白 ;

識別子 = ( 文字 | ギリシャ文字 ),
         { ( 文字 | ギリシャ文字 ) | アラビア数字 | '’' | '_' } ;
```

次の予約前置詞の 1 つと共に始まるすべての識別子は予約されている: init_, inv_, is_, mk_, post_ そして pre_。

型変数識別子 = '@', 識別子 ;

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

記号リテラル = 数値リテラル | ブールリテラル
| nil リテラル | 文字リテラル | テキストリテラル
| 引用リテラル ;

数字 = アラビア数字, { アラビア数字 } ;

数値リテラル = 10 進数値リテラル | 16 進数値リテラル ;

指数 = ('E' | 'e'), ['+' | '-'], 数字 ;

10 進数値リテラル = 数字, ['.', アラビア数字, { アラビア数字 }], [指数] ;

16 進数値リテラル = ('0x' | '0X'), 16 進数字, { 16 進数字 } ;

ブールリテラル = 'true' | 'false' ;

nil リテラル = 'nil' ;

文字リテラル = '', 文字 | エスケープ列
| 多文字, '' ;

エスケープ列 = '\\' | '\r' | '\n' | '\t' | '\f' | '\e' | '\a'
| '\x' 16 進数字,16 進数字 | '\c' 文字
| '\' 8 進数字, 8 進数字, 8 進数字
| '\"' | '\'' | ;

多文字 = ギリシャ文字
| '<=' | '>=' | '<>' | '->' | '+>' | '==>' | '||'
| '=>' | '<=>' | '|->' | '<:' | ':>' | '<-:'
| ':->' | '==' | '**' | '++' ;

テキストリテラル = '"', { '""' | 文字 | エスケープ列 }, '"' ;

引用リテラル = 識別文字,
{ '_' | 識別文字 | アラビア数字 } ;

一行コメント = '--', { 文字 – 改行 }, 改行 ;

上記のエスケープ列は次のように翻訳される:

列	翻訳
'\\'	バックスラッシュ文字
'\r'	リターン文字
'\n'	改行文字
'\t'	タブ文字
'\f'	用紙送り文字
'\e'	エスケープ文字
'\a'	アラーム (ベル)
'\x' 16 進数字,16 進数字	文字の 16 進表示 (たとえば \x41 は 'A')
'\c' 文字	制御文字 (たとえば \c A ≡ \x01)
'\' 8 進数字, 8 進数字, 8 進数字	文字の 8 進表示
'\""'	" 文字
'\''	' 文字

C 演算子優先順位

具象構文における演算子の優先順位は 2 段階で定義される: 演算子はファミリーに区分けされて、上位の優先順位 $>$ がそれらのファミリーに対し与えられるが、その結果ファミリーである F_1 と F_2 が次を満足させるとする

$$F_1 > F_2$$

するとファミリー F_1 のすべての演算子はファミリー F_2 のすべての演算子より高位の優先順をもつ。

ファミリー内での相対的な演算子の優先順は、型情報を考慮して決定され、これが曖昧さの解決に役立つ。型構成子は別に扱われ、他の演算子とともに優先順に並べられることはない。

演算子には 6 つのファミリーすなわち、結合子、適用子、評価子、関係子、連結子、構成子がある:

結合子: 関数値、写像値の結合および、関数値、写像値、数値の反復を許す操作。

適用子: 関数適用、項目選択、列索引、その他、

評価子: 非述語である演算子。

関係子: 関係の演算子。

連結子: 論理連結子。

構成子: 式の構成において、陰に陽に用いられる演算子; たとえば if-then-elseif-else, '|->', '...', その他。

ファミリー上の優先順は次の通り:

結合子 $>$ 適用子 $>$ 評価子 $>$ 関係子 $>$ 連結子 $>$ 構成子

C.1 結合子のファミリー

これらの結合子は最高位の優先順位をもつ。

結合子 = 反復 | 合成 ;

反復 = '**' ;

合成 = 'comp' ;

優先順位	結合子
1	comp
2	iterate

C.2 適用子のファミリー

すべての適用子は等しい優先順位をもつ。

適用子 = 部分列
| 適用
| 関数型インスタンス化
| 項目選択 ;

部分列 = 式, '(', 式, ',', '...', ',',
式, ')' ;

適用 = 式, '(', [式リスト], ')' ;

関数型インスタンス化 = 式, '[', 型, { ',', 型 }, ']' ;

項目選択 = 式, '.', 識別子 ;

C.3 評価子のファミリー

評価子のファミリーは、それらが用いられている式の型に従い、9つのグループに区分けされる。

```
評価子 = 算術前置演算子
       | 集合前置演算子
       | 列前置演算子
       | 写像前置演算子
       | 逆写像
       | 算術中置演算子
       | 集合中置演算子
       | 列中置演算子
       | 写像中置演算子 ;

算術前置演算子 = '+' | '-' | 'abs' | 'floor' ;

集合前置演算子 = 'card' | 'power' | 'dunion' | 'dinter' ;

列前置演算子 = 'hd' | 'tl' | 'len'
             | 'inds' | 'elems' | 'conc' ;

写像前置演算子 = 'dom' | 'rng' | 'merge' | 'inverse' ;

算術中置演算子 = '+' | '-' | '*' | '/' | 'rem' | 'mod' | 'div' ;

集合中置演算子 = 'union' | 'inter' | '\' ;

列中置演算子 = '^' ;

写像中置演算子 = 'munion' | '++' | '<:' | '<-:' | ':>' | ':->' ;
```

優先順位はアナログの演算子のパターンを追いかける。以下の表においてファミリーは定義されている。

優先順位	算術	集合	写像	列
1	+ -	union \	munion ++	^
2	* / rem mod div	inter		
3			inverse	
4			<: <-:	
5			:> :->	
6	(単項) + (単項) - abs floor	card power dinter dunion	dom rng merge	len elems hd tl conc inds

C.4 関係子のファミリー

このファミリーは、結果値が bool 型であるすべての関係演算子を含む。

関係子 = 関係中置演算子 | 集合関係演算子 ;

関係中置演算子 = '=' | '<>' | '<' | '<=' | '>' | '>=' ;

集合関係演算子 = 'subset' | 'psubset' | 'in set' | 'not in set' ;

優先順位	関係子	
1	<=	<
	>=	>
	=	<>
	subset	psubset
	in set	not in set

関係子ファミリーのすべての演算子は等しい優先順位をもつ。タイプしていくということが、これらを隣り合わせに用いるとき意味をもたせる方法のないことを語るものである。

C.5 連結子のファミリー

このファミリーは、結果が bool 型であるすべての論理演算子を含む。

連結子 = 論理前置演算子 | 論理中置演算子 ;

論理前置演算子 = 'not' ;

論理中置演算子 = 'and' | 'or' | '=>' | '<=>' ;

優先順位	連結子
1	<=>
2	=>
3	or
4	and
5	not

C.6 構成子のファミリー

このファミリーは値を構成するために用いられるすべての演算子を含む。これらの優先順は、演算子の暗黙部である括弧によってかまたは構文によって与えられる。

C.7 グループ化

2 項演算子の演算対象のグループ化は以下の通り:

結合子:	右グループ化
適用子:	左グループ化
連結子:	'=>' 演算子は右グループ化 他の演算子は組み合わされるもののため、グループ化の左右は同等である
評価子:	左グループ化[26].
関係子:	グループ化は行わない、意味をなさないからである
構成子:	グループ化は行わない、意味をなさないからである

C.8 型演算子

型演算子は独自の優先順位をもち、以下の通り:

1. 関数型: ->，+> (右グループ化).
2. 合併型: | (左グループ化).
3. 他の 2 項型演算子: * (グループ化なし).
4. 写像型: map ... to ... および inmap ... to ... (右グループ化).
5. 単項型演算子: seq of, seq1 of, set of.

D 2つの具象構文間の相違

以下は数学構文と ASCII 構文との間で異なる記号のリストである:

[26] 右グループ化を行う "写像定義域限定" および "写像定義域削減" 演算子を除く（これは標準ではない）

数学構文	ASCII 構文
$\cdot$	`&`
$\times$	`*`
$\leq$	`<=`
$\geq$	`>=`
$\neq$	`<>`
$\xrightarrow{o}$	`==>`
$\rightarrow$	`->`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\mapsto$	`|->`
$\triangleq$	`==`
$\uparrow$	`**`
$\dagger$	`++`
$\biguplus$	`munion`
$\lhd$	`<:`
$\rhd$	`:>`
$\not\lhd$	`<-:`
$\not\rhd$	`:->`
$\subset$	`psubset`
$\subseteq$	`subset`
$\curvearrowright$	`^`
$\bigcap$	`dinter`
$\bigcup$	`dunion`
$\mathcal{F}$	`power`
...-set	`set of ...`
$\ldots^*$	`seq of ...`
$\ldots^+$	`seq1 of ...`
$\ldots \xrightarrow{m} \ldots$	`map ... to ...`
$\ldots \overset{m}{\longleftrightarrow} \ldots$	`inmap ... to ...`
μ	`mu`
$\mathbb{B}$	`bool`
$\mathbb{N}$	`nat`
$\mathbb{Z}$	`int`
$\mathbb{R}$	`real`
$\neg$	`not`

数学構文	ASCII 構文
$\cap$	`inter`
$\cup$	`union`
$\in$	`in set`
$\notin$	`not in set`
$\wedge$	`and`
$\vee$	`or`
$\forall$	`forall`
$\exists$	`exists`
$\exists!$	`exists1`
λ	`lambda`
ι	`iota`
$\ldots^{-1}$	`inverse ...`

E 標準ライブラリ

E.1 数学ライブラリ

数学ライブラリは `math.vdm` ファイルに定義されている。以下の数学関数を提供する:

関数		事前条件
`sin: real +> real`	Sine	
`cos: real +> real`	Cosine	
`tan: real -> real`	Tangent	引数は $\pi/2$ の整数倍でない
`cot: real -> real`	Cotangent	引数は π の整数倍でない
`asin: real -> real`	Inverse sine	引数は-1 から 1 の間 (両端を含む) にない
`acos: real -> real`	Inverse cosine	引数は-1 から 1 の間 (両端を含む) にない
`atan:real +> real`	Inverse tangent	
`acot:real +> real`	Inverse cotangent	引数は 0 でない
`sqrt: real -> real`	平方根	引数は負でない
`exp: real +> real`	e を底とする指数関数	
`ln: real +> real`	自然対数	引数は正
`log: real +> real`	対数	引数は正
`srand: int ==> ()`	乱数の種 (seed) 初期化	引数は-1 以上
`rand: int ==> int`	擬似乱数生成	引数は 0 より大きい乱数最大値

また次の値が与えられる:

`pi` = 3.14159265358979323846

関数が、ありうる事前条件を満たさないような引数とともに適用される場合に、適当な VDM-SL 値、`Inf` (無限値、たとえば `tan(pi/2)`) および `NaN` (数でない値、たとえば `sqrt (-1)`)、でない値を返すであろう。

モジュールの仕様記述内で標準ライブラリを使用する場合、以下のライブラリファイルを

```
$TOOLBOXHOME/stdlib/math.vdm
```

現在のプロジェクトに追加すべきである。これは MATH モジュールから成り立つ。このライブラリからの関数は、必要に応じてそれらをモジュールに輸入することによって、このライブラリからの関数はいつもの通りにアクセス可能になるだろう。以下の例で紹介する:

```
module UseLib

  imports
    from MATH all

  definitions

  types

  coord :: x : real
           y : real

  functions

  -- euclidean metric between two points
  dist : coord * coord -> real
  dist (c1,c2) ==
    MATH`sqrt((c1.x - c2.x) * (c1.x - c2.x) +
              (c1.y - c2.y) * (c1.y - c2.y));

  -- outputs angle of line joining coord with origin
  -- from horizontal, in degrees
  angle : coord -> real
  angle (c) ==
    MATH`atan (c.y / c.x) * 360 / ( 2 * MATH`pi)

end UseLib
```

E.2 IO ライブラリ

IO ライブラリは io.vdm ファイルで定義され、 $TOOLBOXHOME/stdlib/ ディレクトリに置かれている。以下にリストされた IO 関数と操作が提供される。各々の read/write 関数と操作は、 相当する IO 行為の成功 (true) または失敗 (false) を表すブール値 (またはブール構成要素をもつ組) を返す。

`writeval[@p]:[@p] +> bool`
: この関数は ASCII 形式での VDM 値を標準出力に書き出す。事前条件はない。

`fwriteval[@p]:seq1 of char * @p * filedirective +> bool`
: この関数は ASCII 形式での VDM 値 (第 2 引数) を、第 1 引数に文字列で指定された名称のファイルに書き出す。第 3 パラメーターは以下のように定義された型 filedirective をもつ:

```
filedirective = <start>|<append>
```

: <start> が用いられた場合、存在するファイルは (あった場合には) 上書される; <append> が用いられた場合、出力は存在するファイルの最後に付け足されすでに存在しているものがなければ新しいファイルが生成される。事前条件はない。

`freadval[@p]:seq1 of char +> bool * [@p]`
: この関数は ASCII 形式の VDM 値を、最初の引数の文字列で指定されたファイルから読み取る。事前条件はない。関数は 2 つを返し、第 1 構成要素は読取の成功を示し、第 2 構成要素は読取が成功した場合の読み取られた値を示す。

`echo: seq of char ==> bool`
: この操作は与えられたテキストを標準出力に書き出す。囲みのダブルクォートは取り除かれ、バックスラッシュ文字が escape sequences として翻訳される。事前条件はない。

`fecho: seq of char * seq of char * [filedirective] ==> bool`
: この操作は echo に似ているが、標準出力ではなくファイルに書き出す。filedirective パラメーターは fwriteval に対するものとして翻訳されるべきである。この操作に対する事前条件は、ファイル名称として空列が

与えられた場合、テキストは標準出力に書かれるので [filedirective] 引数は nil であるべきということである。

ferror:() ==> seq of char read/write 関数と操作は、エラーが起きた場合に false を返す。この場合は内部エラー列がセットされる。この操作はこの文字列を返しそれを""にセットする。

IO ライブラリの使用例として、ページヒットのログを保守する web サーバーを考えよう:

```
module LoggingWebServer

  imports
    from IO all

  exports all

  definitions

  values
    logfilename : seq1 of char = "serverlog"

  functions
    URLtoString : URL -> seq of char
    URLtoString = ...

  operations
    RetrieveURL : URL ==> File
    RetrieveURL(url) ==
      (def - = IO`fecho(logfilename, URLtoString(url)^"\n", <append>);
       ...
      );

    ResetLog : () ==> bool
    ResetLog() ==
      IO`fecho(logfilename,"\n",<start>)
```

```
end LoggingWebServer
```

E.3 VDMUtil ライブラリ

VDMUtil ライブラリは、VDMUtil.vdm ファイルに定義され、$TOOLBOXHOME/stdlib/ に配置されている。VDMUtil ライブラリは、以下のリストにあるような VDM のユーティリティを関数や操作として提供する。

`set2seq[@T]:set of @T +> seq of @T`
: この関数は、任意の型を持つ集合を簡単に列に変換することができる。

`get_file_pos: () +> [seq of char * nat * nat * seq of char * seq of char]`
: この関数は、ソースの特定部分のコンテキスト情報（ファイル名、行番号、クラス名、関数・操作名）を抽出することができる。

`P: () +> [seq of char * nat * nat * seq of char * seq of char]`
: この関数は、get_file_pos 関数と同じで、関数名が短い点だけが異なる。

`val2seq_of_char[@T]: @T +> seq of char`
: この関数は、任意の値を文字列に変換することができる。

`seq_of_char2val[@p]:seq1 of char -> bool * [@p]`
: この関数は、文字列を VDM の任意の値に変換することができる。

`cast[@T1, @T2]: @T1 -> [@T2]`
: この関数は、narrow 式と同じ型変換を行う。過去の版との互換性のために存在し、今後、新たに使用する必要はない。

`classname[@T]: @T -> [seq1 of char]`
: インスタンス・オブジェクトのクラス名を得る。

`char2code: char -> nat`
: 文字から、その UTF16 コードの値を得る。

`code2char: nat -> char`
: UTF16 コードの値から、文字を得る。

`current_time: () ==> nat`

1970 年 1 月 1 日からのミリ秒形式で、現在時間を得る。

索 引

abs, 8
and, 5
card, 15
comp
    function composition, 31
    map composition, 21
conc, 18
dinter, 15
div, 8
dom, 21
dunion, 15
elems, 18
floor, 8
hd, 18
in set, 15
inds, 18
inmap to, 20
inter, 15
inverse, 21
len, 18
map to, 20
merge, 21
mk_
    token value, 12
    レコード構成子, 27
    組構成子, 24
mod, 8
munion, 21
not in set, 15
not, 5
or, 5
power, 15
psubset, 15
rng, 21
seq of, 17
seq1 of, 17
set of, 14
subset, 15
tl, 18
union, 15
()
    function apply, 31
    map apply, 21
    sequence apply, 18
**, 21
    function iteration, 31
    numeric power, 8
*, 8
    組型, 24
++
    map override, 21
    sequence modification, 18
+>, 31
+, 8
->, 31
-, 8
.
    record field selector, 27
/, 8
:->, 21
:-, 26
::, 26
:>, 21
<-:, 21
<:, 21
<=>, 5

<=, 8
<>
    boolean inequality, 5
    char inequality, 11
    function inequality, 31
    map inequality, 21
    numeric inequality, 8
    optional inequality, 29
    quote inequality, 12
    record inequality, 27
    sequence inequality, 18
    set inequality, 15
    token inequality, 12
    tuple inequality, 24
    union inequality, 29
    引用値, 11
<, 8
=>, 5
=
    boolean equality, 5
    char equality, 11
    function equality, 31
    map equality, 21
    numeric equality, 8
    optional equality, 29
    quote equality, 12
    record equality, 27
    sequence equality, 18
    set equality, 15
    token equality, 12
    tuple equality, 24
    union equality, 29
>=, 8
>, 8
[]
    optional type, 29
    sequence enumeration, 17
[|]
    sequence comprehension, 17
&
    map comprehension, 20
    sequence comprehension, 17
    set comprehension, 14
\, 15
ˆ, 18
{}
    map enumeration, 20
    set enumeration, 14
{|}
    map comprehension, 20
    set comprehension, 14
bool, 5
char, 11
false, 5
is not yet specified
    functions, 36
    operations, 81
token, 12
true, 5
実数 (real), 7
整数 (int), 7
正の自然数 (nat1), 7
自然数 (nat), 7
10 進数値リテラル, 159
16 進数値リテラル, 159
1 存在限量式, 50, 146
1 対 1 写像型, 20, 134
2 項式, 46, 143
2 項演算子, 47, 143

narrow 式, 63

Absolute value, 8

always 文, 104, 153

Biimplication, 5

call 文, 102, 152
Cardinality, 15
cases 式, 47, 140
cases 式選択肢, 48, 140
cases 式選択肢群, 47, 94, 140
cases 文, 94, 151
cases 文選択肢, 95, 151
cases 文選択肢群, 151
Concatenation, 18
Conjunction, 5
Cosine, 169
Cotangent, 169

dcl 文, 90, 151
def 式, 45, 140
def 文, 89, 150
Difference
    numeric, 8
    set, 15
Disjunction, 5
Distribute merge, 21
Distributed concatenation, 18
Distributed intersection, 15
Distributed union, 15
Division, 8
Domain, 21
Domain restrict by, 21
Domain restrict to, 21

Elements, 18
elseif 式, 47, 140
elseif 文, 94, 151
Equality
    boolean type, 5
    char, 11
    function type, 31
    map type, 21
    numeric type, 8
    optional type, 29
    quote type, 12
    record, 27
    sequence type, 18
    set type, 15
    token type, 12
    tuple, 24
    union type, 29
equality abstraction field, 26
error 文, 108, 153
exit 文, 104, 153

Field select, 27
Finite power set, 15
Floor, 8
for loop, 97
Function apply, 31
Function composition, 31
Function iteration, 31

Greater or equal, 8
Greater than, 8

Head, 18

if 式, 47, 140
if 文, 94, 151
Implication, 5
Indexes, 18
Inequality
    boolean type, 5
    char, 11
    function type, 31
    map type, 21

numeric type, 8
optional type, 29
quote, 12
record, 27
sequence type, 18
set type, 15
token type, 12
tuple, 24
union type, 29
Integer division, 8
Intersection, 15
Inverse cosine, 169
Inverse cotangent, 169
Inverse sine, 169
Inverse tangent, 169
IO, 169, 171
iota 式, 52, 146
is 基本型, 65, 159
is 式, 65, 148

Length, 18
Less or equal, 8
Less than, 8
let be 式, 42, 140
let be 文, 87, 150
let 式, 42, 140
let 文, 87, 150
library, 169

Map apply, 21
Map composition, 21
Map inverse, 21
Map iteration, 21
Math, 169
Membership, 15
Merge, 21
Modulus, 8

narrow 式, 148
Negation, 5
nil リテラル, 159
Not membership, 15

others 式, 48, 140
others 文, 95, 151
Override, 21

pi, 169
Power, 8
Product, 8
Proper subset, 15

Range, 21
Range restrict by, 21
Range restrict to, 21
Remainder, 8
return 文, 103, 152

Sequence application, 18
Sequence modification, 18
Sine, 169
Standard libraries, 169
Subset, 15
Sum, 8

Tail, 18
Tangent, 169
trap 文, 104, 153
trap 群, 104, 153

Unary minus, 8
Union, 15
use シグネチャ, 122, 132

var 情報, 81, 138
VDMUtil, 173

while ループ, 99, 152

より大きい, 144
より大きいか等しい, 144
より小さい, 144
より小さいか等しい, 144
インタフェース, 114, 119, 129
エスケープ列, 159
エラー, 82, 138
エラーリスト, 82, 138
キーワード, 158
テキストリテラル, 160
トークン, 12
パターン, 70, 153
パターンリスト, 36, 71, 81, 154
パターン型ペアリスト, 36, 137
パターン束縛, 70, 155
パターン識別子, 70, 154
パラメーターリスト, 137
パラメーター型, 36, 137
パラメーター群, 36, 81, 137
ブロック文, 90, 150
ブール, 5
ブールリテラル, 159
モジュール, 114, 129
モジュール本体, 114, 132
モード, 82, 138
ラムダ式, 62, 148
レコードパターン, 70, 154
レコード修正, 59, 147
レコード修正子, 59, 147
レコード型, 25, 133
レコード構成子, 59, 147
ローカル定義, 42, 87, 150

一致値, 70, 154
一般 is 式, 65, 148
一般代入文, 92, 151
一般写像型, 20, 134
一行コメント, 160
不変条件, 78, 135
不変条件初期関数, 78, 135
不等, 144
乗算, 144
乱数の種 (seed) 初期化, 169
事前条件式, 68, 148
仕様記述文, 109, 152
代入定義, 91, 151
代入文, 92, 151
任意のモジュール, 111, 129
任意の型, 31, 36, 81, 135
例外, 82, 138

値シグネチャ, 114, 131
値定義, 42, 77, 87, 135
値定義群, 77, 135
値群シグネチャ, 114, 131
値輸入, 119, 130
全称限量式, 50, 146
全関数型, 31, 36, 134
再帰 trap 文, 104, 153
写, 57, 147
写パターン, 70, 154
写パターンリスト, 70, 154
写像中置演算子, 163
写像併合, 145
写像併合パターン, 70, 154
写像修正または列修正, 145
写像値域, 142
写像値域削減, 145
写像値域限定, 145
写像内包, 57, 147
写像列挙, 57, 147
写像列挙パターン, 70, 154
写像前置演算子, 163
写像参照または列参照, 92, 149

写像型, 20, 134
写像定義域, 142
写像定義域削減, 145
写像定義域限定, 145
分配的写像併合, 142
分配的列連結, 142
分配的集合共通部分, 142
分配的集合合併, 142
分離符, 158
列 for ループ, 97, 152
列中置演算子, 163
列先頭, 142
列内包, 55, 146
列列挙, 55, 146
列列挙パターン, 70, 154
列前置演算子, 163
列型, 17, 134
列尾部, 142
列索引, 142
列要素, 142
列連結, 145
列連結パターン, 70, 154
列長, 142
初期化, 78, 135
剰余算, 144
加算, 144
動的リンクインタフェース, 122, 132
動的リンクモジュール, 122, 131
動的リンク輸入型シグネチャ群, 122, 132
動的リンク輸入定義, 122, 132
動的リンク輸入定義リスト, 122, 132
動的リンク輸出シグネチャ, 123, 132
動的リンク輸出定義, 123, 132
包含, 145
単項式, 46, 141
単項演算子, 46, 141
反復, 145, 162
合併型, 29, 134
合成, 145, 162
同値, 145
名称, 66, 149
名称リスト, 66, 82, 149
否定, 141
含意, 144
型, 14, 17, 20, 24, 25, 29, 31, 133
型判定, 65, 148
型名称, 135
型変数, 135
型変数リスト, 36, 136
型変数識別子, 159
型定義, 133
型定義群, 133
型束縛, 62, 76, 155
型束縛リスト, 62, 155
型輸入, 119, 130
型輸出, 114, 131
基本型, 133
外部節, 81, 138
多文字, 160
多重代入文, 92, 151
多重型束縛, 76, 155
多重束縛, 76, 155
多重集合束縛, 76, 155
存在限量式, 50, 146
定義ブロック, 111, 132
対数, 169
帰属, 145
平方根, 169
底値, 141
式, 41, 45–47, 50, 52, 53, 55, 57–60, 62–64, 66, 68, 139

式リスト, 53, 139
引用, 11
引用リテラル, 160
引用型, 133

恒等文, 108, 153
拡張陽操作定義, 81, 138
拡張陽関数定義, 35, 136
括弧型, 133
括弧式, 140
指数, 159
指数関数, 169
接頭辞式, 46, 141
操作シグネチャ, 115, 131
操作型, 81, 138
操作定義, 80, 137
操作定義群, 80, 137
操作本体, 81, 138
操作群シグネチャ, 115, 131
操作輸入, 120, 130
擬似乱数生成, 169
数値リテラル, 159
数字, 159
整数除算, 144
文, 87, 89–91, 94, 96, 99, 100, 102–104, 107–109, 149
文字, 11, 155
文字リテラル, 159
文書, 111, 129
旧名称, 66, 149
有限べき集合, 142
未定義式, 68, 148
束縛, 76, 155
束縛リスト, 50, 76, 155
正符号, 141
法算, 144
減算, 144

状態定義, 78, 135
状態指示子, 92, 149
相等, 144
相等定義, 89, 150
真包含, 145
空列を含まない列型, 17, 134
空列を含む列型, 17, 134
算術中置演算子, 163
算術前置演算子, 163
算術絶対値, 141
索引 for ループ, 97, 152
組パターン, 70, 154
組型, 24, 134
組構成子, 58, 147
組選択, 61, 147
結合子, 162

自然対数, 169
記号リテラル, 159
評価子, 163
論理中置演算子, 165
論理前置演算子, 165
論理和, 144
論理積, 144
識別子, 158
識別子型ペア, 137
識別子型ペアリスト, 36, 137
負符号, 141
輸入シグネチャ, 119, 130
輸入モジュールシグネチャ, 119, 129
輸入値シグネチャ, 119, 130
輸入型シグネチャ, 119, 130
輸入定義, 119, 129
輸入定義リスト, 119, 129
輸入操作シグネチャ, 119, 130
輸入関数シグネチャ, 119, 130
輸出シグネチャ, 114, 131

輸出モジュールシグネチャ, 114, 130
輸出型シグネチャ, 114, 131
輸出定義, 114, 130
輸出関数シグネチャ, 115, 131

逆写像, 46, 142
連結子, 165
適用, 60, 147, 162
適用子, 162
選択型, 29, 134
部分列, 55, 146, 162
部分関数型, 31, 36, 134
関係中置演算子, 164
関係子, 164
関数シグネチャ, 115, 131
関数型, 31, 36, 134
関数型インスタンス化, 61, 147, 162
関数定義, 35, 136
関数定義群, 136
関数本体, 36, 137
関数群シグネチャ, 115, 131
関数輸入, 119, 130
関数輸出, 115, 131
限量式, 50, 146
除算, 144
陰操作定義, 81, 138
陰操作本体, 81, 138
陰関数定義, 35, 136
陽操作定義, 81, 137
陽関数定義, 35, 136
集合 for ループ, 97, 152
集合中置演算子, 163
集合共通部分, 145
集合内包, 54, 146
集合列挙, 53, 146
集合列挙パターン, 70, 154
集合前置演算子, 163
集合合併, 145
集合合併パターン, 70, 154
集合型, 14, 134
集合差, 145
集合束縛, 76, 155
集合濃度, 142
集合範囲式, 54, 146
集合関係演算子, 164
非帰属, 145
非決定文, 100, 152
項目, 25, 134
項目リスト, 25, 133
項目参照, 92, 149
項目選択, 61, 147, 162