Overture Technical Report Series
No. TR-001

September 2016

VDM-10 Language Manual

by

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

Overture – Open-source Tools for Formal Modelling

Document history

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

Contents

Chapter 1

はじめに

The Vienna Development Method (VDM) [**?**, **?**, **?**] は、元々ウィーンの IBM 研究所で 1970 年代に開発され、最も初期に確立された形式手法の一つだ。このドキュメントは、合意された VDM-10 仕様から派生した 3 つの方言、VDM-SL、VDM++、そして VDM-RT を同時に説明する共通言語マニュアルである。これらの方言は VDMTools [**?**]（但し VDM-RT を除く）と、Eclipse プラットフォーム上に構築された Overture オープンソースツール [**?**] の両者でサポートされている。ある特徴が 3 つの方言に共通している場合には「VDM 言語」という表現が用いられるが、特定の方言に固有の特徴を説明している場合には、それがどの方言に対する特徴かを明記する。

1.1 VDM Specification 言語 (VDM-SL)

VDM-SL 言語のシンタックスとセマンティクスは、本質的には ISO 標準の VDM-SL [**?**] にモジュール拡張を加えたものだ[1]。ISO/VDM-SL に準拠したすべての構文的に正しい仕様記述は、VDM-10/VDM-SL としても構文的に正しいことに注目して欲しい。言語に関して明快で理解しやすい説明を行うが、このドキュメント自身は完全な VDM-SL のレファレンスマニュアルではない。言語に対するより完全な解説としては、既存の文献を示す[2]。VDM-10/VDM-SL の記法が ISO 標準/VDM-SL と異なる場合には、もちろん注意深い解説が加えられる。

1.2 VDM++ 言語

VDM++ は、主にエンジニアリング環境で、平行性ならびにリアルタイム性を持つオブジェクト指向システムの記述を行うことが意図された、形式仕様記述言語である [**?**]。この言語は VDM-SL [**?**] に基き、その上に、Smalltalk-80 や Java といった言語の中に見出だせるような、クラスやオブジェクトの概念が拡張されたものだ。

[1] その他の拡張も若干含まれている。
[2] よりチュートリアル風の解説については [**?**, **?**] を参照、VDM-SL を用いた証明に関しては [**?**] と [**?**] が詳しい。

1.3 VDM Real Time 言語 (VDM-RT)

VDM-RT 言語 (以前は VICE「VDM++ In Constrained Environments」と呼ばれていた）は、VDM++ 言語の拡張である。分散したリアルタイム組み込みシステムを適切にモデル化し、分析するために用いられる [**?**, **?**, **?**, **?**, **?**]。

1.4 このドキュメントの目的

このドキュメントは、VDM-10 から派生した全ての方言をカバーした言語レファレンスマニュアルである。VDM 言語の言語要素の構文は文法規則を用いて定義される。それぞれの言語要素の意味はインフォーマルなスタイルで説明され、小さな例題が与えられる。記述は「必要に応じて参照する」スタイルに向いていて、最初から順番に読んでいくことは特に想定されていない。すなわちチュートリアルではなくマニュアルなのである。読者はオブジェクト指向プログラミング/設計の概念をよく知っていることを期待されている。

ドキュメント中では ASCII （交換用）構文を用いるが、全ての予約語は特別なキーワードフォントを用いいて表現する、一般的な Unicode 識別子が許されるので、例えば日本語の文字を直接書くことも可能だ。

1.5 ドキュメントの構造

Chapter 2 では BNF 記法を用いて構文構成要素が示される。VDM の記法に関しては Chapter 3 から Chapter 16で記述される。言語の完全な構文については Appendix A で、字句仕様は Appendix B で、そして演算子の優先度については Appendix C で説明する。Appendix D では数学構文と ASCII 構文で使われるシンボルの違いについての一覧を示した。

Chapter 2

具象構文表記

本書で、言語構文について表現する場合には、派生 BNF 表記を用いる。使用される派生 BNF 表記法には、以下に示す特殊記号が用いられる:

,	連結記号
=	定義記号
|	定義分離記号 (選択肢)
[ ]	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
‘ ’	単引用符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。)

Chapter 3

データ型定義

伝統的なプログラミング言語と同様に、VDM 言語でもデータ型を定義し適切な名称を与えることができる。そうした名前を与えるための等式は以下のような形式になる：

```
Amount = nat
```

ここでは“Amount (合計)”という名のデータ型を定義して、更に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)** は以下説明する基本型の 1 つである)。この時点で述べておく価値のある、VDM 言語の型体系全般に共通する性質の 1 つは、相等と不等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。他の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。しかし VDM 言語には合併型 (後述する) 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 (組込み演算子を用いて) どのように構成され操作され得るのかについて述べる。最初に基本データ型を示し、次に合成型の説明へと進もう。

3.1 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[1]

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。

3.1.1 ブール型

一般的に VDM 言語では、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。このような潜在的な未定義状態を取り扱うために、VDM 言語では 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 ([**?**] 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

[1] 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。

(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[2]:

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)     ≡ ⊥
```

[2]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。

3.1.2 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)**[3]、**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：

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。

[3]VDM 言語 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。

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。

Operator Name	Semantics Description
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 [**?**] 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 \keyw{div} y = -\keyw{floor}(\keyw{abs}(-x/y))
x/y >= 0:   x \keyw{div} y =  \keyw{floor}(\keyw{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 \keyw{rem} y = x - y * (x \keyw{div} y)
x \keyw{mod} y = x - y * \keyw{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
```

3.1.3 文字型

文字型は、VDM 文字集合 (146ページの表 B.1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** → **bool**
c1 <> c2	不等	**char** * **char** → **bool**

例題:

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

3.1.4 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 言語においては、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 → **bool**
q <> r	不等	T * T → **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
```

3.1.5 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 言語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 ISO/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
```

3.2 合成型

以下に合成型について説明する。各々の説明には以下の項目が含まれている：

- 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 [4]
- 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] これらの演算子は、第 6.3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。

3.2.1 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[5]、全体は 1 つとして扱われる。VDM 言語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。

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[6]。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 6.7節に与えられる

演算子:

[5]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 3.2.6節参照)。
[6]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 8節を参照)。

演算子	名称	型
`e` **`in set`** `s1`	帰属	A * **set of** A → **bool**
`e` **`not in set`** `s1`	非帰属	A * **set of** A → **bool**
`s1` **`union`** `s2`	合併	**set of** A * **set of** A → **set of** A
`s1` **`inter`** `s2`	共通部分	**set of** A * **set of** A → **set of** A
`s1 \ s2`	差	**set of** A * **set of** A → **set of** A
`s1` **`subset`** `s2`	包含	**set of** A * **set of** A → **bool**
`s1` **`psubset`** `s2`	真包含	**set of** A * **set of** A → **bool**
`s1 = s2`	相等	**set of** A * **set of** A → **bool**
`s1 <> s2`	不等	**set of** A * **set of** A → **bool**
`card` `s1`	濃度	**set of** A → **nat**
`dunion` `ss`	分配的合併	**set of set of** A → **set of** A
`dinter` `ss`	分配的共通部分	**set of set of** A → **set of** A
`power` `s1`	有限べき集合	**set of** A → **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 3.2.6節に与えられる。

演算子の意味:

Operator Name	Semantics Description
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。
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。

Operator Name	Semantics Description
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}                           ≡ {}
```

3.2.2 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 6.8節で述べる。

演算子:

演算子	名称	型
hd l	先頭	**seq1 of** A → A
tl l	尾部	**seq1 of** A → **seq of** A
len l	長さ	**seq of** A → **nat**
elems l	要素集合	**seq of** A → **set of** A
inds l	索引集合	**seq of** A → **set of nat1**
l1 ^ l2	連結	(**seq of** A) * (**seq of** A) → **seq of** A
conc ll	分配的連結	**seq of seq of** A → **seq of** A
l ++ m	列修正	**seq of** A * **map nat1 to** A → **seq of** A
l(i)	列適用	**seq of** A * **nat1** → A
l1 = l2	相等	(**seq of** A) * (**seq of** A) → **bool**
l1 <> l2	不等	(**seq of** A) * (**seq of** A) → **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 3.2.6節に例題が与えられている。

演算子の意味定義:

Operator Name	Semantics Description
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。

Operator Name	Semantics Description
分配的連結	l1 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>]
```

3.2.3 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 言語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 6.9節で述べる。

演算子:

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

Operator Name	Semantics Description
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。
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> |->                       <Denmark> |-> 4,
       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 }
```

3.2.4 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 6.10節で述べられる。

演算子:

演算子	名称	型
t.#n	選択	T * **nat** → Ti
t1 = t2	相等	T * T → **bool**
t1 <> t2	不等	T * T → **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 6.12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
```

3.2.5 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 言語における **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 3.2.6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 (3.2.6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 6.11節で与えられる。

演算子:

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

演算子の意味定義:

Operator Name	Semantics Description
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。

3.2.6 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 → **bool**
`t1 <> t2`	不等	A * A → **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。

3.2.7 オブジェクト参照型

オブジェクト参照型が標準 VDM-SL の型に加えられた。したがって純粋なオブジェクト指向原則に従うように、オブジェクト参照型 (およびそのオブジェクト) の使用を制限する直接的な方法は存在しない；クラス以外の構造化機能の追加は現段階では予定されていない。こうした原則から考えれば、型構成子 (レコード、写像、集合、等々) と組み合わせられたオブジェクト参照型の使用には、十分な注意を払うべきである。

オブジェクト参照型の値はオブジェクトへの**参照**とみなすことができる。たとえばもし、あるインスタンス変数 (第10.1節参照) がオブジェクト参照型として定義されるならば、このインスタンス変数が定義されているクラスは、オブジェクト参照型の中のクラスの「クライアント」となる；つまり**クライアント関係** がこの 2 つのクラス間に確立する。

オブジェクト参照型はクラス名によって表示される。オブジェクト参照型に含まれるクラス名は、その仕様の中で定義された 1 つのクラスの名称でなければならない。

この型の値に対して定義された演算子は、相等 (‘=’) と不等 (‘<>’) のみである。相等は、値ではなく参照に基づくものとなる。このため、もし `o1` と `o2` が 2 つの異なるオブジェクトならば、たまたま同じ内容であったとしても、`o1 = o2` は false となる。

構成子 オブジェクト参照は new 式 (第 6.13節参照) を用いて構成される。

演算子

演算子	名称	型
`t1 = t2`	相等	A * A → **bool**
`t1 <> t2`	不等	A * A → **bool**

例題 オブジェクト参照の使用例として二分木のクラス定義を示す:

```
class Tree

  types

    \PROTECTED tree = <Empty> | node;

    \PUBLIC node :: lt: Tree
                    nval : int
                    rt : Tree

 instance variables
    \PROTECTED root: tree := <Empty>;
end Tree
```

ここでは node 型を定義している、この型は node の値を nval で、左右の Tree オブジェクトへの参照をそれぞれ lt と rt で行っている。アクセス指定子についての詳細は第**??**節に述べられている。

3.2.8 関数型

VDM 言語では、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 5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`[7] 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 6.16節にある。

演算子:

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

[7] 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 言語においてこれは、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:

Operator Name	Semantics Description
関数適用	関数 `f` を a_j の値に適用した結果。第 6.12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 = \keyw{lambda} x : \keyw{nat} \& \keyw{lambda} y : \keyw{nat} \
f2 = \keyw{lambda} x : \keyw{nat} \& x + 2
inc1 = \keyw{lambda} x : \keyw{nat} \& x + 1
inc2 = \keyw{lambda} y : \keyw{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 言語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。

3.3 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、24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;
```

Chapter 4

アルゴリズム定義

VDM 言語では、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 言語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[1]。VDM 言語では、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。

[1]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。

Chapter 5

関数定義

VDM 言語では、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)、もしく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。

VDM++ と VDM-RT では、関数は自動的に static として提供される（すなわち対象のクラスのインスタンスを作る必要はない）。よって VDM++ と VDM-RT の中の操作で利用するために **`static`** キーワードを使う必要はない。関数はアトミックに実行され、渡されたオブジェクトのインスタンス変数は矛盾なく読み出される。そして関数には sync 句も存在しない（Chapter 14）を参照のこと。関数を定義するための一般的な構文は以下の通り:

```
関数定義 = ‘functions’, [ アクセス関数定義 ],
           { ‘;’ }, アクセス関数定義 関数定義, [ ‘;’ ] ;

アクセス関数定義 = [ アクセス ], 関数定義 ;

アクセス = ‘public’
         | ‘private’
         | ‘protected’ ;

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'
         | 'is subclass responsibility' ;
```

モデルの開発途中では、**is not yet specified**（=「まだ定義されていない」という意味）を関数本体として用いることが許されている。

一方 **is subclass responsibility**（=「サブクラスの責任である」という意味）はこの本体の実装が、このクラスのサブクラスの中で行われなければならいことを示している。よってこの指定は VDM++ と VDM-RT のみで許されている。

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) \verb+\+ (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
```

5.1 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: [1]

```
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:

```
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
```

[1] 多相関数の例は [?] 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。

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。

5.2 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))];
```

よって 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)]\verb+^+(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]

Chapter 6

式

この章では異なる種類の式の 1 つ 1 つについて解説する。各々は次の形式で記述する:

- BNF 構文記法
- 非公式な意味定義記述
- 使用の記述例

6.1 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 5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, \ldots, 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 ISO/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 言語においては、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[1]。 **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 35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 (24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
```

[1]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。

```
    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 (73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 (38ページ) を集合上で考えたものである:

```
    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 \verb+\+ \{x\});
```

別の方法として、この関数を集合内包 (第 6.7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)\verb+^+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]\verb+^+(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。

6.2 def 式

この式は、第 11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:

```
\Keyw{def} pb1 = e1;
    \ldots
    pbn = en
\Keyw{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[2]。

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:

[2]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 7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。

```
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**??**節の操作 GroupRunnerUp_expl でもまた def 式の例が与えられている。

6.3 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 3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 3節で与えられるため、ここでは触れない。

6.4 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 言語における 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 言語における 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]\verb+^+(lmerge (tl s1, s2))
  else [hd s2]\verb+^+(lmerge (s1, tl s2));

mergesort : seq of nat -> seq of nat
mergesort (l) ==
  cases l:
    [] -> [],
    [x] -> [x],
    l1\verb+^+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、26 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"\verb+^+(print_Const(e)),
    mk_Var(id,-) -> "Var of"\verb+^+id,
    mk_Infix(mk\_(e1,op,e2)) -> "Infix of"\verb+^+print_Expr(e1)^","
                                \verb+^+print_Op(op)\verb+^+","
                                \verb+^+print_Expr(e2),
    mk_Cond(t,c,a) -> "Cond of"\verb+^+print_Expr(t)\verb+^+","
                               \verb+^+print_Expr(c)\verb+^+","
                               \verb+^+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。

6.5 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, \ldots, 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, \ldots, 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、[**?**] 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 \verb+\+ \{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
```

6.6 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。

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

6.8 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, \ldots, 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               <Norway>,
GroupA                       <Morocco>]
   & GroupA(i).won <>
0]
[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]
```

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

6.10 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, \ldots, 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
```

6.11 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, \ldots, 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, \ldots, 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)
\MYEQUIV 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 \verb+\+ \{wsc,lsc\}) union \{new_wsc, new_lsc\}
pre forall sc1, sc2 in set gp &
        sc1 <> sc2 <=> sc1.team <> sc2.team
        and \{wt,lt\} subset \{sc.team | sc in set gp\}
```

6.12 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 3.2.5 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 3.2.3 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 3.2.2 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 言語においては、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 (52ページ)、`GroupG` は 1 つの写像 (53ページ)、そして `selection_sort` は 1 つの関数 (42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 5節から例題の関数を用いる:

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

\MYEQUIV

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

6.13 new 式

構文: 式 = ...
| new 式 ;

new 式 = ‘**new**’, 名称, ‘(’, [式リスト], ‘)’ ;

意味定義: *new* 式は次の形式をもつ:

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

new 式を用いることで、クラス記述からオブジェクトを生成すること (これはまた**インスタンス生成**とも呼ばれる) が可能である。*new* 式による効果は、`classname` クラス

に記述された他と識別できる新しいオブジェクトが生成されることである。*new* 式 の値は、新しいオブジェクトへの参照である。

new 式 がパラメーターなしで呼び出された場合は、中のすべてのインスタンス変数は“既定” 値 (i.e. それらの初期化条件で定義された値) をとった 1 つのオブジェクトが生成される。パラメーターありの場合には *new* 式 は **構成子** (第 11.1を参照) に相当し、カスタマイズされたインスタンスを生成する (つまり ここでのインスタンス変数は既定値とは異なる値をとることも許される)。

例題: `Queue` という 1 つのクラスを仮定し、この既定インスタンスは空であるとする。またこのクラスは 1 つの構成子 (これもまた `Queue` と呼ばれる) を含み、これは単一のパラメーターをとりこれが任意のスタートキューを表す値のリストであるとする。このように仮定すると `Queue` の既定インスタンスを生成することができ、実際のキューは次の式を用いて空である

```
new Queue()
```

そして次の式を用いることで `Queue` の 1 つのインスタンスを生成することができるが、 ここにおいて実際のキューはたとえば `e1, e2, e3` となる

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

27 ページで定義されたクラス `Tree` を用いることで、 `nodes` を構成する新しい `Tree` インスタンスを生成する:

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

6.14 self 式

構文:

式 = ...
| self 式 ;

self 式 = ‘**self**’ ;

意味定義: *self* 式は次の形式をもつ:

```
self
```

self 式は現在実行中のオブジェクトへの参照を返す。継承の連鎖における名前空間を単一化するために、用いることができる。

例題: 27 ページで定義された Tree クラス を用いることで、B 木検索アプローチを用いてデータを保存する BST と呼ばれるサブクラスを記述することができる。これにより、B 木検索挿入を実行する操作を指定することができる:

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

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

この操作は、挿入に先立ちそこから行き来するルートを見つけるため、self 式を用いる。更なる例題が第 **??**に示される。

6.15 スレッド ID 式

構文: 式 = ...
| スレッド ID 式 ;

スレッド ID 式 = '**threadid**' ;

意味定義: スレッド *ID* 式は次の形式をもつ:

```
threadid
```

スレッド ID 式は、その式が実行されているスレッドを一意に識別する自然数を返す。周期的なスレッドはそれぞれの周期の状態における新しいスレッド ID を得ることに注意する。

例題: スレッド ID を用いることで、許可述語を使って、VDM++に JAVA スタイルの wait-notify を実装する VDM++基本クラスを提供することが可能となる。この wait-notify 手法で利用できるオブジェクトはすべて、この基本クラスから派生するものでなければならない。

```
class WaitNotify

  instance variables
```

```
    waitset : set of nat := \{\};

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

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

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

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

    \PROTECTED notifyAll: () ==> ()
    notifyAll() ==
         waitset := \{\};

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

\keyw{end} WaitNotify
```

この例ではスレッド ID 式が 2 箇所で用いられている:

- スレッドに対する `Wait` 操作中に、このオブジェクトへの関心を記録するため。
- `Awake` に対する 許可述語中。関与するスレッドは、`Wait` を用いた記録を行った後に `Awake` を呼ぶべきである。そしてこのスレッドは、`notify` に対するもうひとつのスレッド呼出しの後、待ち集合からスレッド ID が取り除かれるまでブロックされる。

周期的なスレッドを持っているときは、wait-notify 構造の使用について注意する必要がある。(なぜなら、それぞれの新しい周期にたいしてスレッド ID が変化するからである)

6.16 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 5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 41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}
```

6.17 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
```

例題 2: この例では、Test() および Test'() の実行結果には差がないが、Test() は型チェックエラーとなる。

```
class S
end S

class C1 is subclass of S

instance variables
public a : nat := 1;

end C1

class C2 is subclass of S

instance variables
public b : nat := 2;

end C2

class A

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

public
```

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

6.18 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: 24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 26にも例題が載せられている。

6.19 基底クラス構成要素

構文: 式 = ...
| isofbaseclass 式
| ... ;

isofbaseclass 式 = '**isofbaseclass**', '(', 識別子, 式, ')' ;

意味定義: 関数 **isofbaseclass** がクラス名である識別子とオブジェクト参照式である式に適用された場合、識別子が上で式で参照されるオブジェクトの、継承チェーン上のルートスーパクラスのとき、かつその時に限り **true** が返される。それ以外の場合には **false** が返される。

例題: BinarySearchTree が Tree のサブクラスであると仮定すると、Tree は他のクラスのサブクラスにはならないし、Queue が Tree や BinarySearchTree に継承によって関係付けられることもない。t を Tree のインスタンス、b を BinarySearchTree のインスタンス、 q を Queue のインスタンス、とすると次の通り:

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

6.20 クラス構成要素

構文 式 = ...
| isofclass 式
| ... ;

isofclass 式 = '**isofclass**', '(', 識別子, 式, ')' ;

意味定義: 関数 **isofclass** が、クラス名を表す識別子とオブジェクト参照である式に適用された場合、式が参照するオブジェクトのクラスが識別子で指定されたものと同じか、またはそのサブクラスである場合に限り、ブール値である **true** が返される。その他の場合には **false** が返される。

例題: 前の例と同様に、Tree,BinarySearchTree, Queue のクラスと、識別子 t,b, q を仮定する:

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

6.21 同基底クラス構成要素

構文:

式 = ...
| samebaseclass 式
| ... ;

samebaseclass 式 = ‘**samebaseclass**’,
‘(’, 式 1, ‘,’, 式 2, ‘)’ ;

意味定義: 関数 **samebaseclass** がオブジェクト参照である式 1 と式 2 に適用された場合、両者が同じルートスーパークラスから派生したクラスのインスタンスであるときに限り、ブール値の **true** が返される。それ以外の場合には **false** が返される。

例題: 前に述べた例題と同様に、クラス Tree、BinarySearchTree, Queue, および識別子 t,b, q を仮定し、AVLTree を Tree のもうひとつのサブクラス、BalancedBST を BinarySearchTree のサブクラス、 a を AVLTree のインスタンス、そして bb を BalancedBST のインスタンスとする:

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

6.22 同クラス構成要素

構文:

式 = ...
| sameclass 式
| ... ;

sameclass 式 = ‘**sameclass**’,
‘(’, 式 1, ‘,’, 式 2, ‘)’ ;

意味定義: 関数 **sameclass** がオブジェクト参照である式 1 と式 2 に適用されると、式 1 と式 2 で参照されるインスタンスのクラスが同じものである場合に限り、ブール値の **true** が返される。それ以外の場合は **false** が返される。

例題: 第 6.19節においてのクラス Tree、BinarySearchTree、Queue、そして識別子 t、b、 q を仮定し、さらに b′ が BinarySearchTree のもうひとつのインスタンスであると仮定した場合、次が得られる:

```
sameclass(b,t)   ≡ false
sameclass(b,b′)  ≡ true
sameclass(q,t)   ≡ false
```

6.23 履歴式

構文:

式 = ...
| act 式
| fin 式
| active 式
| req 式
| waiting 式
| ... ;

act 式 = ‘#act’, ‘(’, 名称, ‘)’
| ‘#act’, ‘(’, 名称リスト, ‘)’ ;

fin 式 = ‘#fin’, ‘(’, 名称, ‘)’
| ‘#fin’, ‘(’, 名称リスト, ‘)’ ;

active 式 = ‘#active’, ‘(’, 名称, ‘)’
| ‘#active’, ‘(’, 名称リスト, ‘)’ ;

req 式 = ‘#req’, ‘(’, 名称, ‘)’
| ‘#req’, ‘(’, 名称リスト, ‘)’ ;

waiting 式 = ‘#waiting’, ‘(’, 名称, ‘)’
| ‘#waiting’, ‘(’, 名称リスト, ‘)’ ;

意味定義: 履歴式は許可述語においてのみ用いられる (第 14.1節参照)。履歴式は以下の式を 1 つ以上含むことが許される:

- #act （操作名）操作名 操作が起動された回数。

- `#fin(操作名)` 操作名操作が完了した回数。
- `#active(操作名)` 現在起動中である操作名操作の数。 このとき: `#active(操作名)` = `#act(操作名)` - `#fin(操作名)`
- `#req(操作名)` 操作名 操作に対して発生した要求の数。
- `#waiting(操作名)` 操作名操作に対する未解決の要求の数。このとき: `#waiting(操作名)` = `#req(操作名)` - `#act(操作名)`

これらすべての演算子に対して、名前リスト版である *#history op(op1,. . .,opN)* は *#history op(op1)* + · · · +*#history op(opN)* に対する簡易な省略形である。

例題: 3 つの操作 A, B そして C が実行されるある特別なスレッドの実行における 1 時点を想定しよう。要求、起動、完了、の列がこのスレッド中で起こる。このことは図6.1において視覚的にみてとれる。

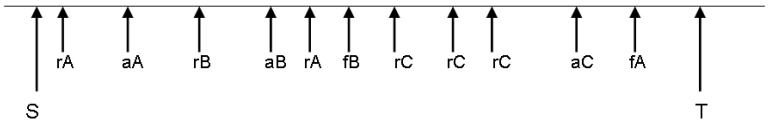

Figure 6.1: 履歴式

ここに記号 *rA* を操作 A の実行要求を示すものとして用い、 *aA* は A の起動を示すもの、*fA* は操作 A の実行の完了を示すものとし、そして同様のことを操作 B と C に対しても用いる。各々の履歴式は、 時間間隔 [**S**,**T**] に対するものとしての以下の値をとる:

```
#act(A) = 1      #act(B) = 1      #act(C) = 1      #act(A,B,C) = 3
#fin(A) = 1      #fin(B) = 1      #fin(C) = 0      #fin(A,B,C) = 2
#active(A) = 0   #active(B) = 0   #active(C) = 1   #active(A,B,C) = 1
#req(A) = 2      #req(B) = 1      #req(C) = 3      #req(A,B,C) = 6
#waiting(A) = 1  #waiting(B) = 0  #waiting(C) = 2  #waiting(A,B,C) = 3
```

6.24 time 式

構文: time 式 = '**time**' ;

意味定義: これは単純に、与えられた CPU 上の現在の時間を問い合わせるものである。時間は自然数で提供されます。

例: たとえば、確実にある操作が実行されたことをログ（記録）に取りたい場合、`logEnvToSys` のようにして操作を作成することができる。

```
public logEnvToSys: nat ==> ()

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

6.25 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モジュール名/クラス名をを示している (第 **??**節並びに第 **??**節および付録 Bも参照)。旧名称 は、操作定義の事後条件 (第 11節参照) および仕様文の事後条件 (第 **??**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)。

VDM-SL における旧名称の使用例として、以下のように定義された state を考えてみよう:

```
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
```

また VDM++/VDM-RT における旧名称の使用例としては、以下のように定義されたインスタンス変数を考えてみよう:

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

これによって変数 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 11節の操作についてを参照)。

モジュール/クラス名の簡単な例として、ここで以下のように、 build_rel という関数が CGRel というモジュール/クラス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) == {|->}
```

別のモジュール/クラスにおいては、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にアクセスすることができる

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

なお VDM++ と VDM-RT では build_rel 関数はアクセスモディファイアを追加することによって定義されているクラスの外からのアクセスを許すことができる。

6.26 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
```

6.27 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。

Chapter 7

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 **elems**GroupA 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}
```

46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。

Chapter 8

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);
```

Chapter 9

値(定数)定義

値 (value) 定義は、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 = '**values**', [アクセス値定義],
{ '**;**', アクセス値定義 }, ['**;**'] ;

アクセス値定義 = [アクセス], 値定義 ;

値定義 = パターン, ['**:**', 型], '=', 式 ;

意味定義: 値定義は以下の形式を持つ

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

なお `access` パートは VDM++ と VDM-RT でのみ利用可能である。

(値定義で定義された) グローバル値は、 VDM 言語記述の全レベルで参照が可能である。しかしながら、1 つの仕様記述が実行可能であるためには、使用される前に値定義の中で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、VDM インタープリタによって値定義にのみ適用される。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 ISO/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。値定義にも他と同様に型制限を提供することが可能なので、そ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。

VDM++ および VDM-RT のアクセス記述子の詳細は、第 **??**節で述べられる。

例題: 以下の例は [**?**] 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

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

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。アクセスモディファイアの**private**、**protected**、**public**はVDM++とVDM-RTでのみ利用可能である。トップレベルのVDM-SL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で成り立つことができる(節**??**参照)。値定義の順番が重要であるように、ある値定義を他のモジュールの定義に依存させないことが推奨される。

Chapter 10

変更可能な状態要素の宣言

VDM の操作を用いて値を変更することのできる状態要素の定義は、VDM-SL と VDM++（VDM-RT）では構文的に異なっている。VDM-SL はモジュールベースであるため、状態定義は単一のレコード型のような構成要素となる。一方、 VDM++ と VDM-RT はオブジェクト指向ベースであるため状態要素は定義の継承を可能にするためにより柔軟でなければならず、このためインスタンス変数という形式で定義される。本章の以下の 2 つの節では、それぞれの状態定義方法について説明を行う。

10.1 インスタンス変数 (VDM++ and VDM-RT)

クラス定義からインスタンス化されたオブジェクトと、そのクラス自身のどもらも、それぞれ内部状態を持つことができる。それはオブジェクトまたはクラスの **インスタンス変数** とも呼ばれる。オブジェクトの場合には、この状態をオブジェクト内のグローバル状態として参照する。

構文: インスタンス変数定義群 = ‘**instance**’, ‘**variables**’,
[インスタンス変数定義,
{ ‘**;**’, インスタンス変数定義 }] ;

インスタンス変数定義 = アクセス指定定義
| 不変条件定義 ;

アクセス指定定義 = ([アクセス], [‘**static**’]) | ([‘**static**’], [アクセス]),
代入定義 ;

代入定義 = 識別子, ‘**:**’, 型, [‘**:=**’, 式] ;

不変条件定義 = ‘**inv**’, 式 ;

意味定義: 内部状態を定義するセクションを始めるためには、`instance variables` というキーワードが必要である。このキーワードに続けてインスタンス変数定義と不変条

件定義のリストが並べられる。各々のインスタンス変数定義は、対応する型指定をともなうインスタンス変数名からなる。そこには初期値とアクセスおよび **static** 指定子を含めることもできる。アクセスおよび **static** 指定子についての詳細は、第**??**節にみることができる。

不変条件定義の方法により、インスタンス変数の値を制限することが可能である。各々の不変条件定義は、1 つ以上のインスタンス変数を含み、クラスオブジェクトのインスタンス変数の値上で定義される可能性もある。スーパークラスから継承されるものを含むクラスにあるすべてのインスタンス変数は、不変条件式で使用することができる。各不変条件定義は、式が true となるようにインスタンス変数の値を制限するブール式でなければならない。すべての不変条件式は、そのクラスの各々のオブジェクトの全存在期間で true となる必要がある。

あるクラスの全てにわたる不変条件式というのは、そのクラスおよびその複数のスーパークラスの不変条件定義のすべてが、まずは 1) 複数のスーパークラス、次は 2) そのクラス自身、の中で定義された順に論理 **and** で結合されたものである。

この操作はプライベートのものであり、パラメーターはもたず、不変条件式の実行に対応し 1 つのブール値を返す。

例題: 以下の例はインスタンス変数定義を示している。このクラスでは 1 つのインスタンス変数が詳しく述べられる:

```
\keyw{class} GroupPhase

\keyw{types}

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


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

end GroupPhase
```

10.2 状態定義 (VDM-SL)

仕様記述内でグローバル変数が必要な場合には、状態定義をすることが可能である。この状態定義の要素は、その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。モジュールの状態定義は、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 VDM 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。

Chapter 11

操作定義

操作については第 4節ですでに述べてきた。ここでは一般的な形式について説明し、 VDM++ と VDM-RT で、クラスのインスタンスの構成に用いられる、**構成子** という名前の特別な操作に関しては、第 11.1節で説明を行う。なお **async** というキーワードは、VDM-RT だけで用いることができる。

構文:

操作定義群 = ‘**operations**’, [アクセス操作定義],
{ ‘**;**’, アクセス操作定義 } , [‘**;**’] ;

アクセス操作定義 = ([‘**pure**’], [‘**async**’] [アクセス], [‘**static**’])
| ([‘**pure**’], [‘**async**’] [‘**static**’], [アクセス]),
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 y = 文
| '**is not yet specified**'
| '**is subclass responsibility**' ;

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: VDM での操作はデフォルトでは同期であるが、キーワード "**async**" が VDM-RT の操作定義の前で使用されると、その操作は非同期操作として扱われることを意味する。このことが意味するのは、その操作は戻り値を持つことができず、非同期操作を呼び出したスレッドは自身の実行をそのまま続行するということである。なおコンストラクタは非同期であると宣言できないことに注意して欲しい。

もしある操作が "**pure**" であると宣言されていて、その操作が関数の文脈で使用された場合（すなわち関数、不変条件、事前/事後条件の中で使用された場合）には、その実行はアトミックに行われる。関数の文脈でない場合に呼び出されたときは、pure 操作の呼び出しは基本的に通常の操作と同じであるが、幾つかの制約が課されている。以下にその制約を挙げる ;

- pure 操作は状態を変えることはできない
- pure 操作は非 pure 操作を呼ぶことはできない
- pure 操作には permission predicates 定義することはできない
- pure 操作を再定義する操作も、また pure 操作でなければならない
- pure 操作に History counters を使うことはできない
- mutex は pure 操作を参照することはできない
- pure 操作は値を返さなければならない
- pure 操作は **async** として宣言することはできない。なぜなら非同期操作は戻り値の型として **void** を持つことを要求されているからだ
- スレッドの本体は pure 操作にはなれない
- pure 操作の中で **exit** を呼び出すことはできない

VDM++ と VDM-RT の両者の、アクセスと **static** 指定子の詳細については、第 **??** 節で解説される。静的操作の中では、静的でない操作を呼ぶことは許されていない、また self 式を静的操作の定義内で用いることはできない、ということも注意しよう。

以下は陽操作の例である。1 つのチームがもうひとつを打ち負かす場合に VDM-SL なら状態 `GroupPhase` を、VDM++ なら `GroupPhase` クラスのインスタンス変数を更新する様子を示すものだ。

```
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 12節で述べられている。文は、必要とするどのような状態/ インスタンス変数に対しても、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)}};
```

外部節では、その操作が扱う状態/インスタンス変数をリストする。予約語である **rd** の後にリストされた状態/インスタンス変数は読み取りのみができるが、**wr** の後にリストされた変数は読みとりと書きだしの両方行うことができる。

VDM-SL の場合、事前、事後条件の定義が存在すると、インタープリタも操作定義の事前、事後条件に対応した新しい関数を作成する。しかし、もし仕様記述がグローバルな状態を含んでいた場合、その状態も新たに作成された関数の一部となる。したがって、以下のシグネチャを持つ関数が、事前、事後条件を伴う操作に対して作成される。[1]:

```
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。

[1] しかしながら、これらの事前、事後条件の述語は、単なるブール関数であることを忘れてはならない。また、その状態は、このような述語関数を呼び出すことによって変化しない

- 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 は旧状態（操作呼び出し前の状態）であり、2 番目の State 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:

```
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。例外節が存在する理由は、正常なケースと例外ケースを切り離すことを可能にするためである。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[2]。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

[2] 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。

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。

errs 句を用いる操作定義は、オリジナルの事前条件と c1,..., cn の全ての条件の論理和が実質的な事前条件となる。また、この場合の実質的な事後条件は、(orig_pre **and** orig_post) と c1 **and** r1,..., cn **and** rn の論理和となる。

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。

次に示す例の VDM-SL の操作は、以下の状態定義を使用するものとする:

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

また VDM++/VDM++ を使っている場合には、次のインスタンス変数定義を使用するものとする:

```
instance variables
  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ている:

```
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。ここでの例外は、キューが空の場合で、そのときに操作がどのように振る舞うべきかを指定している。

VDM-SL モデルに対する VDM インタープリターは以下の関数を作成することに留意する:

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

11.1 構成子 (VDM++ and VDM-RT)

構成子は操作である。それ自身が定義されたクラスと同じ名前をもち、そのクラスの新しいインスタンスを作る。このため戻り値の型は、その同じクラス名でなければならず、もし戻り値が指定されるとすれば **self** となるべきであるが、これは省略可能で、暗黙的に **self** が返される。そして、構成子はクラスの新しいインスタンスを初期化するために使用されるので、構成子を **static** と宣言することは許されていない。

1 つのクラス内では、第 **??**節で述べられる操作のオーバーローディングを用いて多重構成子を定義することが可能である。

Chapter 12

文

Chapter 13

VDMにおける仕様のトップレベル

Chapter 14

同期と制約 (VDM++ and VDM-RT)

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

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

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

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

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

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

synchronization = permission predicates ;

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

14.1 Permission Predicates

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

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

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

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

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

per *operation name* => *guard condition*

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

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

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

Guard conditions can be conceptually divided into:

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

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

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

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

would be translated to the following permission predicates:

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

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

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

14.1.1 History guards

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

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

#act: operation name $\to \mathbb{N}$
#fin: operation name $\to \mathbb{N}$

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

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

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

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

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

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

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

14.1.2 The object state guard

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

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

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

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

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

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

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

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

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

14.1.3 Queue condition guards

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

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

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

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

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

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

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

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

would always block.

14.1.4 Evaluation of Guards

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

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

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

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

14.2 Inheritance of Synchronization Constraints

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

14.2.1 Mutex constraints

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

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

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

```
class A
operations

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

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

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

class B is subclass of A
operations

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

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

end B
```

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

The actual permission predicates in the derived class are therefore:

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

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

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

Chapter 15

スレッド (VDM++ and VDM-RT)

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

Syntax: thread definitions = '**thread**', [thread definition] ;

thread definition = periodic thread definition
| procedural thread definition ;

periodic thread definition = periodic obligation
| sporadic obligation ;

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

15.1 Periodic Thread Definitions (VDM-RT)

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

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

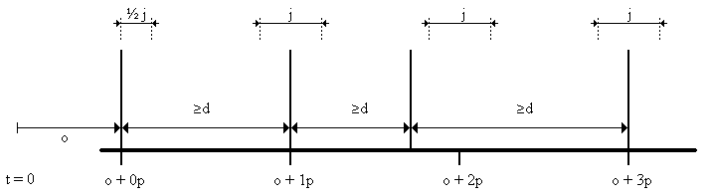

Figure 15.1: Period (p), jitter (j), delay (d) and offset (o)

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

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

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

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

Periodic obligations are intended to describe e.g. analogue physical relations between values in formulas (e.g. transfer functions) and their discrete event simulation. It is a requirement

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

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

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

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

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

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

The Timer provides straightforward operations which need no further explanation.

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

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

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

  IncTime: () ==> ()
  IncTime() ==
    if acti
```

```
    then curTime := curTime + 100;
```

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

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

end Timer
```

15.2 Sporadic Thread Definitions (VDM-RT)

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

Syntax: (VDM-RT):

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

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

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

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

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

```
class Timer
...

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

end Timer
```

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

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

Syntax: procedural thread definition = statement ;

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

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

```
class Factorial

instance variables
  result : nat := 5;
operations

public factorial : nat ==> nat
factorial(n) ==
  if n = 0
  then return 1
  else (dcl m : Multiplier;
        m := new Multiplier();
        m.calculate(1,n);
        start(m);
        result:= m.giveResult();
        return result
```


```
      )

end Factorial

class Multiplier

instance variables
  i : nat1;
  j : nat1;
  k : nat1;
  result : nat1

operations

public calculate : nat1 * nat1 ==> ()
calculate (first, last) ==
  (i := first; j := last);

doit : () ==> ()
doit() ==
( if i = j
  then result := i
  else (dcl p : Multiplier;
        dcl q : Multiplier;
        p := new Multiplier();
        q := new Multiplier();
        start(p);
        start(q);
        k := (i + j) div 2;
        -- division with rounding down
        p.calculate(i,k);
        q.calculate(k+1,j);
        result := p.giveResult() *  q.giveResult ()
       )
);

public giveResult : () ==> nat1
giveResult() ==
  return result;

sync
-- cyclic constraints allowing only the
```

```
-- sequence calculate; doit; giveResult

per doit => #fin (calculate) > #act(doit);
per giveResult => #fin (doit) > #act (giveResult);
per calculate => #fin (giveResult) = #act (calculate)

thread
  doit();

end Multiplier
```

Chapter 16

トレース定義

In order to automate the testing process VDM-10 contains a notation enabling the expression of the traces that one would like to have tested exhaustively. Such traces are used to express combinations of sequences of operations that wish to be tested in all possible combinations. In a sense this is similar to model checking limitations except that this is done with real and not symbolic values. However, errors in test cases are filtered away so other test cases with the same prefix will be skipped automatically.

Syntax: traces definitions = '**traces**', [named trace], { '**;**', named trace } ;

named trace = identifier, { '/', identifier }, '**:**', trace definition list ;

trace definition list = trace definition term, { '**;**', trace definition term } ;

trace definition term = trace definition
| trace definition term, '|', trace definition ;

trace definition = trace binding definition
| trace repeat definition ;

trace binding definition = trace let def binding
| trace let best binding ;

trace let def binding = '**let**', local definition, { '**,**', local definition },
'**in**', trace definition ;

trace let best binding = '**let**' multiple bind, ['**be**', '**st**', expression],
'**in**', trace definition ;

trace repeat definition = trace core definition, [trace repeat pattern] ;

```
trace repeat pattern = '*'
                     | '+'
                     | '?'
                     | '{', numeric literal, '}'
                     | '{', numeric literal, ',' numeric literal, '}' ;

trace core definition = trace apply expression
                      | trace concurrent expression
                      | trace bracketed expression ;

trace apply expression = call statement ;

trace concurrent expression = '||', '(', trace definition,
                              ',', trace definition,
                              { ',', trace definition }, ')' ;

trace bracketed expression = '(', trace definition list, ')' ;
```

Semantics: Semantically the trace definitions provided in a class have no effect. These definitions are simply used to enhance testing of a VDM specification using principles from combinatorial testing (also called all-pairs testing). So each trace definition can be considered as a regular expression describing the test sequences in which different operations should be executed to test the VDM specification. Inside the trace definitions, bindings may appear and for each possible binding a particular test case can be automatically derived. So one trace definition expands into a set of test cases. In this sense a test case is a sequence of operation calls executed after each other. Between each test case the VDM specification is initialised so they become entirely independent. From a static semantics perspective it is important to note that the expressions used inside trace definitions must be executed in the expansion process. This means that it cannot directly refer to instance variables, because these could be changed during the execution.

So here it makes sense to explain what kind of expansion the different kinds of trace definitions gives rise to.

The *trace definition lists* simply use a semicolon ("**;**") and this results in sequencing between the *trace definition terms* used inside it.

In the *trace definition term* it is possible to introduce alternatives using the bar ("|") operator. This results in test cases for all alternatives.

The *trace binding definition* exists in two forms where the *trace let def binding* simply enables the binding introduced to be used after the '**in**' in the same way as in let-expressions. Alternatively the *trace let best binding* can be used and this will expand to test cases with all the different possible bindings.

The *trace repeat definition* is used to introduce the possibility of having repetitions of the operation calls used in the trace. The different kinds of repeat patterns have the following meanings:

- '*' means 0 to n occurences (n is tool specific).
- '+' means 1 to n occurences (n is tool specific).
- '?' means 0 or 1 occurences.
- '{', n, '}' means n occurences.
- '{', n, ',' m '}' means between n and m occurences.

The *trace core definitions* have three possibilities. These are ordinary operation calls, trace concurrency expressions and bracketed trace definitions respectively. The *trace concurrency expressions* are similar to the *nondeterministic statements* in the sense that the trace definition lists inside it will be executed in all possible permulations of the elements. This is particular useful for concurrent VDM++ models where potential deadlocks can occur under some circumstances.

Examples: In an example like the one below test cases will be generated in all possible combination starting with a call of `Reset` followed by one to four `Push`es of values onto the stack followed again by one to three `Pop`s from the stack.

```
class Stack

instance variables
    stack : seq of int := [];

operations

    public Reset : () ==> ()
    Reset () ==
      stack := [];

    public Pop : () ==> int
    Pop() ==
      def res = hd stack in
        (stack := tl stack;
         return res)
    pre stack <> []
    post stack~ = [RESULT] ^ stack;

    public Push: int ==> ()
```

```
    Push(elem) ==
      stack := stack ^ [elem];

    public Top : () ==> int
    Top() ==
      return (hd stack);

end Stack
class UseStack

instance variables

  s : Stack := new Stack();

traces

  PushBeforePop : s.Reset();
                  (let x in set {1,2} in s.Push(x)){1,4};
                  s.Pop(){1,3}

end UseStack
```

Appendix A

The Syntax of the VDM Languages

This appendix specifies the complete syntax for the VDM languages.

A.1 VDM-SL Document

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

any module = module ;

A.1.1 Modules

This entire subsection is not present in the current version of the VDM-SL standard.

module = '**module**', identifier, interface,
[module body], '**end**', identifier ;

interface = [import definition list],
export definition ;

import definition list = '**imports**', import definition,
{ ',', import definition } ;

import definition = '**from**', identifier, import module signature ;

import module signature = '**all**'
| import signature, { import signature } ;

import signature = import types signature
| import values signature
| import functions signature
| import operations signature ;

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

type import = name, ['**renamed**', name]
| type definition, ['**renamed**', name] ;

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

value import = name, ['**:**', type], ['**renamed**', name] ;

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

function import = name, [[type variable list], '**:**', function type],
['**renamed**', name] ;

import operations signature = '**operations**', operation import,
{ '**;**', operation import }, ['**;**'] ;

operation import = name, ['**:**', operation type], ['**renamed**', name] ;

export definition = '**exports**', export module signature ;

export module signature = '**all**'
| export signature,
{ export signature } ;

export signature = export types signature
| export values signature
| export functions signature
| export operations signature ;

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

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

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

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

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

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

export operations signature = '**operations**' operation signature,
{ '**;**', operation signature }, ['**;**'] ;

operation signature = name list, '**:**', operation type ;

A.2 VDM++ and VDM-RT Document

document = class | system , { class | system } ;

A.3 System (VDM-RT)

system = '**system**', identifier,
[class body],
'**end**', identifier ;

A.3.1 Classes

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

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

A.4 Definitions

class body = definition block, { definition block } ;

module body = '**definitions**', definition block, { definition block } ;

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

A.4.1 Type Definitions

type definitions = '**types**', [access type definition] ,
{ '**;**', access type definition }, ['**;**'] ;

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

The access part is only possible in VDM++ and VDM-RT.

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

type definition = identifier, '=', type, [ invariant ]
                | identifier, '::', field list, [ invariant ] ;

type = bracketed type
     | basic type
     | quote type
     | composite type
     | union type
     | product type
     | optional type
     | set type
     | seq type
     | map type
     | partial function type
     | type name
     | type variable ;

bracketed type = '(', type, ')' ;

basic type = 'bool' | 'nat' | 'nat1' | 'int' | 'rat'
           | 'real' | 'char' | 'token' ;

quote type = quote literal ;

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

field list = { field } ;

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

union type = type, '|', type, { '|', type } ;

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

optional type = '[', type, ']' ;

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

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

set1 type = '**set1 of**', type ;

seq type = seq0 type
| seq1 type ;

seq0 type = '**seq of**', type ;

seq1 type = '**seq1 of**', type ;

map type = general map type
| injective map type ;

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

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

function type = partial function type
| total function type ;

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

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

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

type name = name ;

type variable = type variable identifier ;

invariant = '**inv**', invariant initial function ;

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

A.4.2 The VDM-SL State Definition

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

invariant = '**inv**', invariant initial function ;

initialisation = '**init**', invariant initial function ;

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

A.4.3 Value Definitions

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

access value definition = [access], value definition ;

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

A.4.4 Function Definitions

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

access function definition = [access], function definition ;

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

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

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

In VDM-SL extended explicit function definition looks like:

extended explicit function definition = identifier, [type variable list],
parameter types,
identifier type pair list,
'==', function body,
['**pre**', expression],
['**post**', expression] ;

In VDM++ and VDM-RT extended explicit function definition looks like:

extended explicit function definition = identifier, [type variable list],
parameter types,
identifier type pair list,
'==', function body,
['**pre**', expression],
['**post**', expression] ;

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

identifier type pair = identifier, ':', type ;

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

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

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

parameters list = parameters, { parameters } ;

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

function body = expression
| '**is subclass responsibility**'
| '**is not yet specified**' ;

A.4.5 Operation Definitions

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

access operation definition = (['**pure**'], ['**async**'] [access], ['**static**'])
| (['**pure**'], ['**async**'] ['**static**'], [access]),
operation definition ;

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

explicit operation definition = identifier, ':', operation type,
identifier, parameters,
'==', operation body,
['**pre**', expression],
['**post**', expression] ;

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

implicit operation body = [externals],
['**pre**', expression],
'**post**', expression,
[exceptions] ;

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

operation type = discretionary type, '==>', discretionary type ;

operation body = statement
| '**is subclass responsibility**'
| '**is not yet specified**' ;

externals = '**ext**', var information, { var information } ;

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

mode = '**rd**' | '**wr**' ;

exceptions = '**errs**', error list ;

error list = error, { error } ;

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

A.4.6 Instance Variable Definitions (VDM++ and VDM-RT)

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

instance variable definition = access assignment definition
| invariant definition ;

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

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

A.4.7 Synchronization Definitions (VDM++ and VDM-RT)

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

synchronization = permission predicates ;

permission predicates = permission predicate,
{ ';', permission predicate } ;

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

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

A.4.8 Thread Definitions (VDM++ and VDM-RT)

thread definitions = '**thread**', [thread definition] ;

thread definition = periodic thread definition
| procedural thread definition ;

periodic thread definition = periodic obligation
| sporadic obligation ;

For VDM-RT where time is explicit, it looks like:

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

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

For both VDM++ and VDM-RT, we can define:

procedural thread definition = statement ;

A.4.9 Trace Definitions

traces definitions = '**traces**', [named trace], { ';', named trace } ;

named trace = identifier, { '/', identifier }, ':', trace definition list ;

trace definition list = trace definition term, { ';', trace definition term } ;

trace definition term = trace definition
| trace definition term, '|', trace definition ;

trace definition = trace binding definition
| trace repeat definition ;

trace binding definition = trace let def binding
| trace let best binding ;

;

trace let def binding = '**let**', local definition, { ',', local definition },
'**in**', trace definition ;

trace let best binding = '**let**' multiple bind, ['**be**', '**st**', expression],
'**in**', trace definition ;

trace repeat definition = trace core definition, [trace repeat pattern] ;

trace repeat pattern = '*'
| '+'
| '?'
| '{', numeric literal, '}'
| '{', numeric literal, ',' numeric literal, '}' ;

trace core definition = trace apply expression
| trace concurrent expression
| trace bracketed expression ;

trace apply expression = call statement ;

trace concurrent expression = '||', '(', trace definition,
',', trace definition,
{ ',', trace definition }, ')' ;

trace bracketed expression = '(', trace definition list, ')' ;

A.5 Expressions

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

expression = bracketed expression
| let expression
| let be expression
| def expression
| if expression
| cases expression
| unary expression
| binary expression
| quantified expression
| iota expression
| set enumeration
| set comprehension
| set range expression
| sequence enumeration
| sequence comprehension
| subsequence
| map enumeration
| map comprehension
| tuple constructor
| record constructor
| record modifier
| apply
| field select
| tuple select
| function type instantiation
| lambda expression
| new expression
| self expression
| threadid expression
| general is expression
| undefined expression
| precondition expression
| isofbaseclass expression
| isofclass expression
| samebaseclass expression
| sameclass expression
| act expression
| fin expression
| active expression

| req expression
| waiting expression
| time expression
| name
| old name
| symbolic literal ;

A.5.1 Bracketed Expressions

bracketed expression = '(', expression, ')' ;

A.5.2 Local Binding Expressions

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

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

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

A.5.3 Conditional Expressions

if expression = '**if**', expression, '**then**', expression,
{ elseif expression },
'**else**', expression ;

elseif expression = '**elseif**', expression, '**then**', expression ;

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

cases expression alternatives = cases expression alternative,
{ ',', cases expression alternative } ;

cases expression alternative = pattern list, '->', expression ;

others expression = '**others**', '->', expression ;

A.5.4 Unary Expressions

unary expression = prefix expression
| map inverse ;

prefix expression = unary operator, expression ;

unary operator = unary plus
| unary minus
| arithmetic abs
| floor
| not
| set cardinality
| finite power set
| distributed set union
| distributed set intersection
| sequence head
| sequence tail
| sequence length
| sequence elements
| sequence indices
| sequence reverse
| distributed sequence concatenation
| map domain
| map range
| distributed map merge ;

unary plus = '+' ;

unary minus = '−' ;

arithmetic abs = '**abs**' ;

floor = '**floor**' ;

not = '**not**' ;

set cardinality = '**card**' ;

finite power set = '**power**' ;

distributed set union = '**dunion**' ;

distributed set intersection = '**dinter**' ;

sequence head = '**hd**' ;

sequence tail = '**tl**' ;

sequence length = '**len**' ;

sequence elements = '**elems**' ;

sequence indices = '**inds**' ;

sequence reverse = '**reverse**' ;

distributed sequence concatenation = '**conc**' ;

map domain = '**dom**' ;

map range = '**rng**' ;

distributed map merge = '**merge**' ;

map inverse = '**inverse**', expression ;

A.5.5 Binary Expressions

binary expression = expression, binary operator, expression ;

binary operator = arithmetic plus
| arithmetic minus
| arithmetic multiplication
| arithmetic divide
| arithmetic integer division
| arithmetic rem
| arithmetic mod
| less than
| less than or equal
| greater than
| greater than or equal
| equal
| not equal
| or
| and
| imply
| logical equivalence
| in set

| not in set
| subset
| proper subset
| set union
| set difference
| set intersection
| sequence concatenate
| map or sequence modify
| map merge
| map domain restrict to
| map domain restrict by
| map range restrict to
| map range restrict by
| composition
| iterate ;

arithmetic plus = '+' ;

arithmetic minus = '-' ;

arithmetic multiplication = '*' ;

arithmetic divide = '/' ;

arithmetic integer division = '**div**' ;

arithmetic rem = '**rem**' ;

arithmetic mod = '**mod**' ;

less than = '<' ;

less than or equal = '<=' ;

greater than = '>' ;

greater than or equal = '>=' ;

equal = '=' ;

not equal = '<>' ;

or = '**or**' ;

and = '**and**' ;

imply = '=>' ;

logical equivalence = '<=>' ;

in set = '**in set**' ;

not in set = '**not in set**' ;

subset = '**subset**' ;

proper subset = '**psubset**' ;

set union = '**union**' ;

set difference = '\' ;

set intersection = '**inter**' ;

sequence concatenate = '^' ;

map or sequence modify = '++' ;

map merge = '**munion**' ;

map domain restrict to = '<:' ;

map domain restrict by = '<-:' ;

map range restrict to = ':>' ;

map range restrict by = ':->' ;

composition = '**comp**' ;

iterate = '**' ;

A.5.6 Quantified Expressions

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

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

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

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

A.5.7 The Iota Expression

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

A.5.8 Set Expressions

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

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

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

A.5.9 Sequence Expressions

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

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

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

A.5.10 Map Expressions

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

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

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

A.5.11 The Tuple Constructor Expression

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

A.5.12 Record Expressions

record constructor = '**mk_**',[1] name, '(', [expression list], ')' ;

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

record modification = identifier, '|->', expression ;

A.5.13 Apply Expressions

apply = expression, '(', [expression list], ')' ;

field select = expression, '.', identifier ;

tuple select = expression, '.#', numeral ;

function type instantiation = name, '[', type, { ',', type }, ']' ;

A.5.14 The Lambda Expression

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

A.5.15 The narrow Expression

narrow expression = '**narrow_**', '(', expression, ',', type, ')' ;

A.5.16 The New Expression (VDM++ and VDM-RT)

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

A.5.17 The Self Expression (VDM++ and VDM-RT)

self expression = '**self**' ;

A.5.18 The Threadid Expression (VDM++ and VDM-RT)

threadid expression = '**threadid**' ;

[1] **Note:** no delimiter is allowed

A.5.19 The Is Expression

general is expression = is expression
| type judgement ;

is expression = '**is_**',[2] name, '(', expression, ')'
| is basic type, '(', expression, ')' ;

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

A.5.20 The Undefined Expression

undefined expression = '**undefined**' ;

A.5.21 The Precondition Expression

pre-condition expression = '**pre_**', '(', expression,
[{ ',', expression }], ')' ;

A.5.22 Base Class Membership (VDM++ and VDM-RT)

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

A.5.23 Class Membership (VDM++ and VDM-RT)

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

A.5.24 Same Base Class Membership (VDM++ and VDM-RT)

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

A.5.25 Same Class Membership (VDM++ and VDM-RT)

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

[2] **Note:** no delimiter is allowed

A.5.26 History Expressions (VDM++ and VDM-RT)

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

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

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

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

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

A.5.27 Time Expressions (VDM-RT)

time expression = '**time**' ;

A.5.28 Names

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

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

old name = identifier, '~' ;

A.6 State Designators

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

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

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

A.7 Statements

statement = let statement
| let be statement
| def statement
| block statement
| general assign statement
| if statement
| cases statement
| sequence for loop
| set for loop
| index for loop
| while loop
| nondeterministic statement
| call statement
| specification statement
| start statement
| start list statement
| stop statement
| stop list statement
| duration statement
| cycles statement
| return statement
| always statement
| trap statement
| recursive trap statement
| exit statement
| error statement
| identity statement ;

A.7.1 Local Binding Statements

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

local definition = value definition
| function definition ;

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

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

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

A.7.2 Block and Assignment Statements

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

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

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

general assign statement = assign statement
| multiple assign statement ;

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

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

A.7.3 Conditional Statements

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

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

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

cases statement alternatives = cases statement alternative,
{ ',', cases statement alternative } ;

cases statement alternative = pattern list, '->', statement ;

others statement = **'others'**, '->', statement ;

A.7.4 Loop Statements

sequence for loop = '**for**', pattern bind, '**in**',
expression, '**do**', statement ;

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

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

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

A.7.5 The Nondeterministic Statement

nondeterministic statement = '||', '(', statement,
{ ',', statement }, ')' ;

A.7.6 Call and Return Statements

In VDM-SL a call statement looks like:

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

In VDM++ and VDM-RT a call statement looks like:

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

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

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

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

return statement = '**return**', [expression] ;

A.7.7 The Specification Statement

specification statement = '[', implicit operation body, ']' ;

A.7.8 Start and Start List Statements (VDM++ and VDM-RT)

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

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

A.7.9 Stop and Stop List Statements (VDM++ and VDM-RT)

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

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

A.7.10 The Duration and Cycles Statements (VDM-RT)

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

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

A.7.11 Exception Handling Statements

always statement = '**always**', statement, '**in**', statement ;

trap statement = '**trap**', pattern bind, '**with**', statement,
'**in**', statement ;

recursive trap statement = '**tixe**', traps, '**in**', statement ;

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

exit statement = '**exit**', [expression] ;

A.7.12 The Error Statement

error statement = '**error**' ;

A.7.13 The Identity Statement

identity statement = '**skip**' ;

A.8 Patterns and Bindings

A.8.1 Patterns

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

pattern identifier = identifier | '-' ;

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

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

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

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

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

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

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

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

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

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

record pattern = 'mk_',[3] name, '(', [pattern list], ')' ;

object pattern = '**obj_**', identifier, '(', [field pattern list], ')'[4];

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

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

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

[3] **Note:** no delimiter is allowed
[4] **Note:** object pattern is only be used in VDM++ and VDM-RT

A.8.2 Bindings

pattern bind = pattern | bind ;

bind = set bind | seq bind | type bind ;

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

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

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

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

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

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

multiple seq bind = pattern list, '**in seq**', expression ;

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

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

Appendix B

Lexical Specification

B.1 Characters

The characters that comprise a valid VDM specification are defined in terms of Unicode codepoints. The actual character encoding of a VDM source file (for example UTF-8, ISO-Latin-1 or Shift-JIS) is not defined, and the tool support is responsible for converting whatever encoding is used into Unicode during the parse of the file.

All VDM keywords and delimiter tokens are composed of characters from the Basic Latin block ("ASCII" codepoints less than U+0080). On the other hand, user identifiers (variable names, function names and so on) can be composed of a rich variety of Unicode codepoints, reflecting the need for fully internationalized specifications.

All Unicode codepoints have a "category". Certain categories are entirely excluded from the set of codepoints that are permitted in identifiers. This prevents, say, punctuation characters from being used. On the other hand, to provide a degree of compatibility with the original VDM ISO standard, and for backward compatibility, there are different rules for the formation of user identifiers that only use ASCII characters. For example, the underscore is permitted in identifiers (U+005F), even though this is in the connecting punctuation category, which would not normally be allowed.

See `http://www.fileformat.info/info/unicode/category/index.htm` for more information about categories.

initial letter:
if codepoint < U+0100
then Any character in categories Ll, Lm, Lo, Lt, Lu or U+0024 (a dollar sign)
else Any character except categories Cc, Zl, Zp, Zs, Cs, Cn, Nd, Pc

following letter:
if codepoint < U+0100
then Any character in categories Ll, Lm, Lo, Lt, Lu, Nd or U+0024 (a dollar sign)
or U+005F (underscore) or U+0027 (apostrophe)
else Any character except categories Cc, Zl, Zp, Zs, Cs, Cn

digit:

```
0   1   2   3   4   5   6   7   8   9
```

hexadecimal digit:

```
0   1   2   3   4   5   6   7   8   9
A   B   C   D   E   F
a   b   c   d   e   f
```

octal digit:

```
0   1   2   3   4   5   6   7
```

Table B.1: Character set

B.2 Symbols

The following kinds of symbols exist: keywords, delimiters, symbolic literals, and comments. The transformation from characters to symbols is given by the following rules; these use the same notation as the syntax definition but differ in meaning in that no separators may appear between adjacent terminals. Where ambiguity is possible otherwise, two consecutive symbols must be separated by a separator.

keyword = '**#act**' | '**#active**' | '**#fin**' | '**#req**' | '**#waiting**' | '**abs**'
| '**all**' | '**always**' | '**and**' | '**as**' | '**async**' | '**atomic**' | '**be**'
| '**bool**' | '**by**' | '**card**' | '**cases**' | '**char**' | '**class**'
| '**comp**' | '**compose**' | '**conc**' | '**cycles**' | '**dcl**' | '**def**'
| '**definitions**' | '**dinter**' | '**div**' | '**dlmodule**' | '**do**'
| '**dom**' | '**dunion**' | '**duration**' | '**elems**' | '**else**' | '**elseif**'
| '**end**' | '**error**' | '**errs**' | '**exists**' | '**exists1**' | '**exit**'
| '**exports**' | '**ext**' | '**false**' | '**floor**'
| '**for**' | '**forall**' | '**from**' | '**functions**' | '**hd**' | '**if**' | '**in**'
| '**inds**' | '**inmap**' | '**instance**' | '**int**' | '**inter**'
| '**imports**' | '**init**' | '**inv**' | '**inverse**' | '**iota**' | '**is**'
| '**isofbaseclass**' | '**isofclass**' | '**lambda**' | '**len**' | '**let**'
| '**map**' | '**measure**' | '**merge**' | '**mod**' | '**module**' | '**mu**'
| '**munion**' | '**mutex**' | '**nat**' | '**nat1**' | '**new**' | '**nil**' | '**not**' | '**of**'
| '**operations**' | '**or**' | '**others**' | '**per**' | '**periodic**' | '**post**'
| '**power**' | '**pre**' | '**private**' | '**protected**' | '**psubset**'
| '**public**' | '**pure**' | '**rat**' | '**rd**' | '**real**' | '**rem**' | '**renamed**'
| '**responsibility**' | '**return**' | '**reverse**' | '**rng**'
| '**samebaseclass**' | '**sameclass**' | '**self**' | '**seq**' | '**seq1**'
| '**set**' | '**set1**' | '**skip**' | '**specified**' | '**sporadic**' | '**st**' | '**start**'
| '**startlist**' | '**state**' | '**stop**' | '**stoplist**'
| '**struct**' | '**subclass**' | '**subset**' | '**sync**'
| '**system**' | '**then**' | '**thread**' | '**threadid**' | '**time**' | '**tixe**'
| '**tl**' | '**to**' | '**token**' | '**traces**' | '**trap**' | '**true**' | '**types**'
| '**undefined**' | '**union**' | '**uselib**' | '**values**'
| '**variables**' | '**while**' | '**with**' | '**wr**' | '**yet**' | '**RESULT**' ;

identifier = initial letter, { following letter } ;

Note that in VDM-RT the `CPU` and `BUS` classes are reserved and cannot be redefined by the user. These two predefined classes contain the functionality described in Section **??** above.

All identifiers beginning with one of the reserved prefixes are reserved: **`init_`**, **`inv_`**, **`is_`**, **`mk_`**, **`post_`** and **`pre_`**.

type variable identifier = '`@`', identifier ;

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

symbolic literal = numeric literal | boolean literal
| nil literal | character literal | text literal
| quote literal ;

numeral = digit, { digit } ;

numeric literal = decimal literal | hexadecimal literal ;

exponent = (‘E’ | ‘e’), [‘+’ | ‘−’], numeral ;

decimal literal = numeral, [‘.’, digit, { digit }], [exponent] ;

hexadecimal literal = (‘0x’ | ‘0X’), hexadecimal digit, { hexadecimal digit } ;

boolean literal = ‘**true**’ | ‘**false**’ ;

nil literal = ‘**nil**’ ;

character literal = ‘ ’ ’, character | escape sequence
| ‘’’ ;

escape sequence = ‘\\’ | ‘\r’ | ‘\n’ | ‘\t’ | ‘\f’ | ‘\e’ | ‘\a’
| ‘\x’ hexadecimal digit,hexadecimal digit
| ‘\u’ hexadecimal digit,hexadecimal digit,
hexadecimal digit,hexadecimal digit
| ‘\c’ character
| ‘\’ octal digit, octal digit, octal digit
| ‘\"’ | ‘\’’ | ;

text literal = ‘ " ’, { ‘ \" ’ | character | escape sequence }, ‘ " ’ ;

quote literal = ‘<’, identifier, ‘>’ ;

Single-line comment = ‘−−’, { character – newline }, newline ;

Multiple-line comment = ‘/∗’, { character }, ‘∗/’ ;

The escape sequences given above are to be interpreted as follows:

Sequence	Interpretation
'\\'	U+005C (backslash character)
'\r'	U+000D (return character)
'\n'	U+000A (newline character)
'\t'	U+0009 (tab character)
'\f'	U+000C (formfeed character)
'\e'	U+001B (escape character)
'\a'	U+0007 (alarm (bell))
'\x' hexadecimal digit, hexadecimal digit	U+00xy (hex representation of character (e.g. \x41 is 'A'))
'\u' hexadecimal digit, hexadecimal digit, hexadecimal digit, hexadecimal digit	U+abcd (hex representation of character (e.g. \u0041 is 'A'))
'\c' character	U+00nn (control character) (e.g. \cA ≡ \x01)
'\' octal digit, octal digit, octal digit	U+00nn (octal representation of character)
'\"'	U+0022 (double quote)
'\''	U+0027 (apostrophe)

Table B.2: Escape sequences

Appendix C

Operator Precedence

The precedence ordering for operators in the concrete syntax is defined using a two-level approach: operators are divided into families, and an upper-level precedence ordering, $>$, is given for the families, such that if families F_1 and F_2 satisfy

$$F_1 > F_2$$

then every operator in the family F_1 is of a higher precedence than every operator in the family F_2.

The relative precedences of the operators within families is determined by considering type information, and this is used to resolve ambiguity. The type constructors are treated separately, and are not placed in a precedence ordering with the other operators.

There are six families of operators, namely Combinators, Applicators, Evaluators, Relations, Connectives and Constructors:

Combinators: Operations that allow function and mapping values to be combined, and function, mapping and numeric values to be iterated.

Applicators: Function application, field selection, sequence indexing, etc.

Evaluators: Operators that are non-predicates.

Relations: Operators that are relations.

Connectives: The logical connectives.

Constructors: Operators that are used, implicitly or explicitly, in the construction of expressions; e.g. **`if-then-elseif-else`**, '| ->', '...', etc.

The precedence ordering on the families is:

combinators $>$ applicators $>$ evaluators $>$ relations $>$ connectives $>$ constructors

C.1 The Family of Combinators

These combinators have the highest family priority.

combinator = iterate | composition ;

iterate = '**' ;

composition = '**comp**' ;

precedence level	combinator
1	**comp**
2	**iterate**

C.2 The Family of Applicators

All applicators have equal precedence.

applicator = subsequence
| apply
| function type instantiation
| field select ;

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

apply = expression, '(', [expression list], ')' ;

function type instantiation = expression, '[', type, { ',', type }, ']' ;

field select = expression, '.', identifier ;

C.3 The Family of Evaluators

The family of evaluators is divided into nine groups, according to the type of expression they are used in.

evaluator = arithmetic prefix operator
| set prefix operator
| sequence prefix operator
| map prefix operator
| arithmetic infix operator
| set infix operator
| sequence infix operator
| map infix operator ;

arithmetic prefix operator = '+' | '−' | '**abs**' | '**floor**' ;

set prefix operator = '**card**' | '**power**' | '**dunion**' | '**dinter**' ;

sequence prefix operator = '**hd**' | '**tl**' | '**len**'
| '**inds**' | '**elems**' | '**conc**' | '**reverse**' ;

map prefix operator = '**dom**' | '**rng**' | '**merge**' | '**inverse**' ;

arithmetic infix operator = '+' | '−' | '*' | '/' | '**rem**' | '**mod**' | '**div**' ;

set infix operator = '**union**' | '**inter**' | '\' ;

sequence infix operator = '^' ;

map infix operator = '**munion**' | '++' | '<:' | '<−:' | ':>' | ':−>' ;

The precedence ordering follows a pattern of analogous operators. The family is defined in the following table.

precedence level	arithmetic	set	map	sequence
1	+ −	**union** \	**munion** ++	^
2	* / **rem** **mod** **div**	**inter**		
3			**inverse**	
4			<: <−:	
5			:> :−>	
6	(unary) + (unary) − **abs** **floor**	**card** **power** **dinter** **dunion**	**dom** **rng** **merge**	**len** **elems** **hd** **tl** **conc** **inds** **reverse**

Table C.1: Operator precedence

C.4 The Family of Relations

This family includes all the relational operators whose results are of type **bool**.

relation = relational infix operator | set relational operator ;

relational infix operator = '=' | '<>' | '<' | '<=' | '>' | '>=' ;

set relational operator = '**subset**' | '**psubset**' | '**in set**' | '**not in set**' ;

precedence level	relation	
1	<=	<
	>=	>
	=	<>
	subset	**psubset**
	in set	**not in set**

All operators in the Relations family have equal precedence. Typing dictates that there is no meaningful way of using them adjacently.

C.5 The Family of Connectives

This family includes all the logical operators whose result is of type **bool**.

connective = logical prefix operator | logical infix operator ;

logical prefix operator = '**not**' ;

logical infix operator = '**and**' | '**or**' | '=>' | '<=>' ;

precedence level	connective
1	<=>
2	=>
3	**or**
4	**and**
5	**not**

C.6 The Family of Constructors

This family includes all the operators used to construct a value. Their priority is given either by brackets, which are an implicit part of the operator, or by the syntax.

C.7 Grouping

The grouping of operands of the binary operators are as follows:

Combinators: Right grouping.

Applicators: Left grouping.

Connectives: The '=>' operator has right grouping. The other operators are associative and therefore right and left grouping are equivalent.

Evaluators: Left grouping[1].

Relations: No grouping, as it has no meaning.

Constructors: No grouping, as it has no meaning.

C.8 The Type Operators

Type operators have their own separate precedence ordering, as follows:

1. Function types: `->, +>` (right grouping).
2. Union type: `|` (left grouping).
3. Other binary type operators: `*` (no grouping).
4. Map types: **`map`** ... **`to`** ... and **`inmap`** ... **`to`** ... (right grouping).
5. Unary type operators: **`seq of`**, **`seq1 of`**, **`set of`**, **`set1 of`**.

[1] Except the "map domain restrict to" and the "map domain restrict by" operators which have a right grouping. This is not standard.

Appendix D

Differences between the Concrete Syntaxes

When VDM was originally developed a mathematical syntax was used and this have also been retained in the ISO/VDM-SL standard. However, most VDM tools today mainly use an ASCII syntax. Below is a list of the symbols which are different in the mathematical syntax and the ASCII syntax:

Mathematical syntax	ASCII syntax
$\cdot$	`&`
$\times$	`*`
$\leq$	`<=`
$\geq$	`>=`
$\neq$	`<>`
$\xrightarrow{o}$	`==>`
$\rightarrow$	`->`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\mapsto$	`|->`
$\triangleq$	`==`
$\uparrow$	`**`
$\dagger$	`++`
$\biguplus$	**`munion`**
$\lhd$	`<:`
$\rhd$	`:>`
$\not\lhd$	`<-:`
$\not\rhd$	`:->`
$\subset$	**`psubset`**
$\subseteq$	**`subset`**
$\frown$	`^`
$\bigcap$	**`dinter`**
$\bigcup$	**`dunion`**

Mathematical syntax	ASCII syntax
$\mathcal{F}$	`power`
...-set	`set of ...`
...-set$_1$	`set1 of ...`
...*	`seq of ...`
...$^+$	`seq1 of ...`
...$\xrightarrow{m}$...	`map ... to ...`
...$\overset{m}{\longleftrightarrow}$...	`inmap ... to ...`
μ	`mu`
$\mathbb{B}$	`bool`
$\mathbb{N}$	`nat`
$\mathbb{N}_1$	`nat1`
$\mathbb{Z}$	`int`
$\mathbb{R}$	`real`
$\neg$	`not`
$\cap$	`inter`
$\cup$	`union`
$\in$	`in set`
$\notin$	`not in set`
$\wedge$	`and`
$\vee$	`or`
$\forall$	`forall`
$\exists$	`exists`
$\exists!$	`exists1`
λ	`lambda`
ι	`iota`
...$^{-1}$	`inverse ...`