VDM、SPINから始める形式手法入門
第2回：VDM++入門

国立情報学研究所　石川　冬樹
f-ishikawa@nii.ac.jp
2011年8月9日

セミナー日程

8/8（月）	***形式手法概論***	そもそも「形式手法」とは何かを知り， 基本的な考え方を学ぶ
8/9（火）	***VDM++入門***	代表的な手法・ツールを実際に触ってみて，形式手法に共通する原則・基本的な考え方を学ぶ．また，記述や検証に関する異なるアプローチに応じた効果や，その有効範囲について考えてみる
8/10（水）	***SPIN入門***	
8/11（木）	***形式手法ツール概論***	学んだ手法・ツールとの比較を通して 他の代表的な手法・ツールにおける 異なるアプローチを紹介，俯瞰する
8/12（金）	***応用事例紹介***	近年盛んに発表されている，応用事例集や 導入ガイダンスなどについて解説する

本日の内容

- VDMの概要紹介，簡単な演習
  - 簡単な演習を通し，プログラミング言語にはない抽象的な記述を行ってみる
  - 簡単な例題を通し，形式仕様全体の記述の考え方を学んでみる

目次

- VDM概要（手法・ツール）
- 抽象的・宣言的な構文の活用
- 全体の形式仕様記述
- 一般論・補足

VDM

- VDM: Vienna Development Method（手法名）
  - 1970年代にIBMウィーン研究所にて開発
  - モデル指向形式仕様記述：以下の側面を記述
    - システムの状態（変数），それらの不変条件
    - データ型に対する演算（関数），状態を読み書きする機能（操作），それらの事前条件・事後条件
    - モジュール（クラス）の構造
  - ライトウェイトな手法と呼ばれる
    - 実行による比較的手軽な検証・妥当性確認
  - Felicaの適用事例が有名

VDMにおける形式言語

- VDMにおける形式言語
  - CやJavaに近くプログラマに馴染みやすい言語だが，詳細を省いたより抽象的な記述を行う
  - VDM-SL（言語名）
    - 1970年代にIBMのウィーン研究所が開発
    - 1996年にBase Language部分がISO標準に
  - VDM++（言語名）
    - オブジェクト指向を取り込んだ文法
    - 「スレッド制御」等の概念も導入
    - （実行時間を扱う拡張も）

VDM Tools

- デンマークのIFAD社 → CSK
- 一通りのOSに対応
  - Windows, Mac(Intel/G5/G4), Linux
- 異なるバージョン
  - VDM-SL Toolbox: VDM-SL用
  - VDM++ Toolbox: VDM++用
  - VDM++ VICE Toolbox: リアルタイム性導入

VDM Tools

- 基礎編ではVDM-SL Toolboxを利用
  - 現在バージョン8.3.1（2011/05/27リリース）
  - シンクライアントにはAcademic版
  - 自身のマシンにインストールしたい方は，Webサイトに登録するとLite版がダウンロード可能
    （コード生成など一部機能を除いて利用可能）

http://www.vdmtools.jp/

VDM Tools

- VDM Toolsの機能（一部はVDM++のみ）
  - 構文チェック，型チェック
  - 証明課題生成
  - インタプリタ実行（条件の動的チェック），デバッガ
  - コードカバレッジ計測
  - プログラムコードへの変換（C++，Java）
  - プログラムコードからの変換（Java）
  - CORBA APIによる外部プログラムとの連携
  - UMLツールとの連携

Overture IDE

- Eclipseベースのオープンソースツール
  - 2011年バージョン1.0リリース
  - VDM Toolsにない機能
    - エディタ（構文色分け，オンライン構文チェック，キーワード補完，テンプレート挿入）
    - 正規表現のようなパターンに基づいた網羅的なテストケースの記述，実行，結果閲覧
    - その他証明ツールなど開発中

http://www.overturetool.org/

ツール利用について

- 今回は時間節約のためVDMToolsのみを利用
- 基本的な機能はどちらでも同じ
  - エディタはOverture IDEがよい
    - 最初はオンライン文法チェック必須
  - VDMToolsの方が安定（UMLリンクやコード生成などの発展的機能利用時，日本語利用時？）
  - あとは好み

ツールの利用手順は下記にも掲載

http://research.nii.ac.jp/~f-ishikawa/vdm/tools.html

目次

- VDM概要（手法・ツール）
- **抽象的・宣言的な構文の活用**
- 全体の形式仕様記述
- 一般論・補足

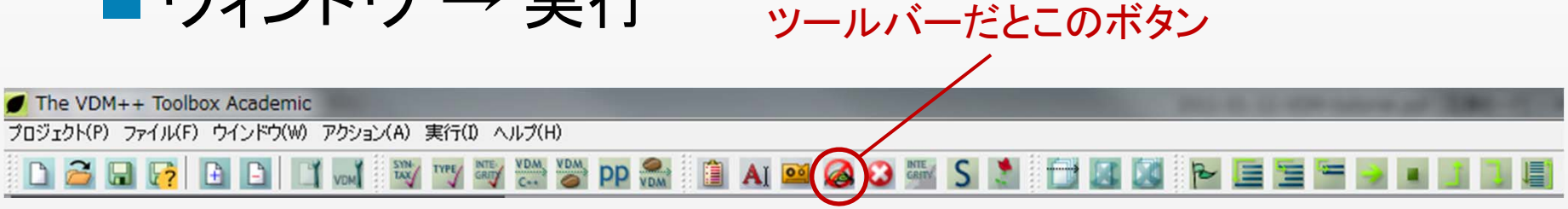
VDM++ Toolbox利用の流れ

- ツール利用の一般的な流れ
  - ファイル上にVDM++記述を行う
  - ツールを用いて文法チェック・型チェックを行う
  - ツールを用いて実行，検証する
  - カバレッジレポート機能やデバッグ機能も用いて，結果の分析を行い，必要に応じ修正し繰り返す

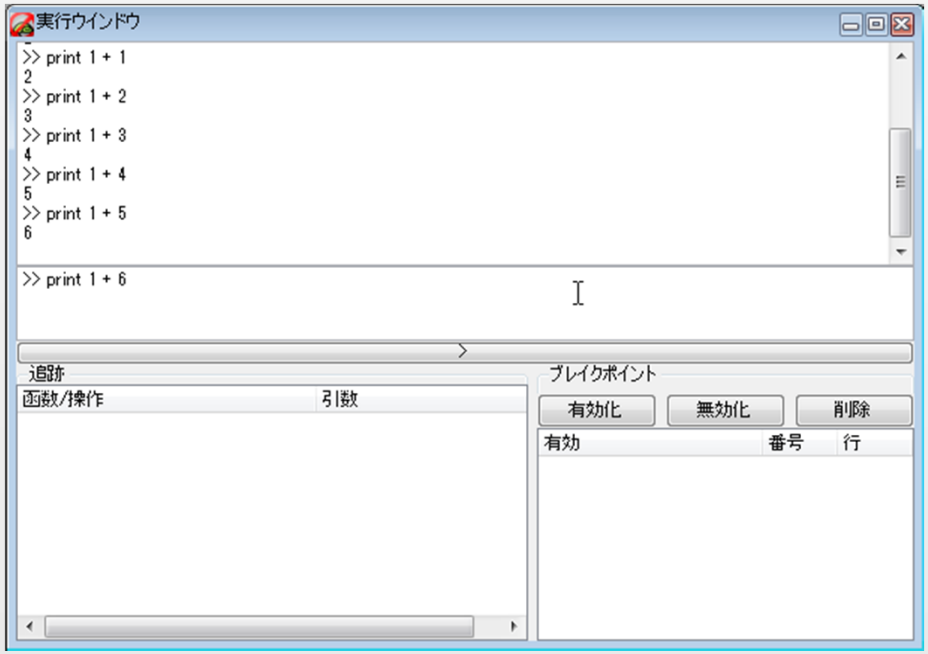
しかしまずは，インタプリタを用いて1つの式を実行，抽象的な構文（代表として集合型）に慣れる

インタプリタ画面表示

- ウィンドウ → 実行



- 実行ウィンドウが開く



インタプリタの基本的な利用法(１)

- 一般的なシェル(コマンドプロンプト)に近い
  - 上下キー(↑, ↓)によりコマンド履歴を参照可能
  - Toolboxにて定義されたコマンドを用いる
  - printコマンドに続けてVDM++の式を与えることにより，その式を評価した結果を表示

    print 1 + 1　（と打ってEnter）

    ➡ 2 と出力される

インタプリタの実行例

- 基本データ型に関する演算式を出力
  （リファレンス・マニュアル参照）
  - bool型
    - print 3 > 1 and 1 > 2
    - print 3 > 1 or 1 > 2
  - 数値型（real, int, nat, nat1）
    - print floor 2.5
    - print 13 div 5
    - print 13 mod 5
    - print 2 ** 3

集合型

- ある型の値を，順序・重複を気にせずに保持するもの
- 型定義の例
  - set of int, set of char, set of set of nat, ...
- 値の例
  - 外延型定義：要素を列挙
    - {1, 3, 5}, {'a', 's', 't'}, {"Japan", "France"}, { }, ...
  - 内包型定義：要素の満たす性質を定義（後述）

集合型

- 等しいかどうかの判定
  - 順序・重複を気にしない　例：　{1, 3, 5} = {1, 5, 3, 1}
  - 参照型ではなく値型：値そのものを比較

    （VDM++では全般に「等しい」判定は = 記号，「等しくない」判定は <> 記号）
- 演算子（次スライド，マニュアル・リファレンス）
  - 集合論における演算を直接表現している
    - 和集合（∩），積集合（∪），帰属関係（∈），包含関係（⊂），べき集合など

集合型

■ 演算子

e in set *s1*	*e*が*s1*の要素である	3 in set {1,2,3} = true
e not in set *s1*	*e*が*s1*の要素ではない	3 not in set {1,2,3} = false
s1 union *s2*	*s1*と*s2*の和集合	{1,2,3} union {2,4} = {1,2,3,4}
s1 inter *s2*	*s1*と*s2*の共通部分	{1,2,3} inter {2,4} = {2}
s1 ￥ *s2*	*s1*と*s2*の差集合 （*s2*の要素を*s1*から除いたもの）	{1,2,3} ￥ {2,4} = {1,3}
s1 subset *s2*	*s1*が*s2*の部分集合（等しくてもよい）	{2,3} subset {1,2,3} = true
s1 psubset *s2*	*s1*が*s2*の真部分集合（等しくはない）	{2,3} psubset {2,3} = false
card *s1*	*s1*の濃度（要素の個数）	card {2,5,6} = 3
dunion *ss*	集合の集合*ss*内の全要素の和集合	dunion {{1,2,3},{2,4}} = {1,2,3,4}
dinter *ss*	集合の集合*ss*内の全要素の共通部分	dinter {{1,2,3},{2,4}} = {2}
power *s1*	集合*s1*のべき集合	power {1,2} = {{},{1},{2},{1,2}}

集合の内包型定義

- 内包型定義: 集合の要素の満たす性質を指定
  - { x * x | x in set {1, 2, 3, 4} & x mod 2 = 0}
    - 集合 {1, 2, 3, 4} の要素をそれぞれ抜き出して x と呼んだとして,
    - x を 2 で割った余りが 0 になるもののみ抜き出し, (この場合 2 と 4)
    - それらの x の値に対してそれぞれ x * x を計算したものを集めた集合を求める (この場合 {4, 16})

集合の内包型定義

- 内包型定義の例
  - { x | x in set {1, 2, 3, 4} & x mod 2 = 0} (= {2, 4})
  - { x + 3 | x in set {1, 2, 3, 4} & x mod 2 = 0}
    (= {5, 7})
  - { x * y | x in set {1, 2, 3, 4}, y in set {1, 2, 3, 4}
    & x mod 2 = 0 and y mod 2 = 1}
    (= { 2 * 1, 4 * 1, 2 * 3, 4 * 3} = {2, 4, 6, 12})
  - { x * y | x, y in set {1, 2, 3, 4}
    & x mod 2 = 0 and y mod 2 = 1}
    （結果は上と同じ）

演習問題1－1

- 以下の式の評価結果を予測し，実際にインタプリタ上で評価（printコマンド）してみて確認してみよ
  - { x | x in set {1, ..., 6} & x * x > 20}
  - { x + y | x, y in set {1, 2, 3, 4}
    & x mod 2 = 0 and y mod 2 = 1}
  - { x | x in set power {1, 2, 3, 4} & card x = 2 }
    （集合に関する演算子参照）

集合の内包型定義

- 内包型定義の利用イメージ
  - 電車の座席集合から空席を抜き出す

```
{ x | x in set getAllSeats(trainID)
    & not isReserved(x) }
```

  - チケット予約記録の集合から，あるユーザが予約を行ったものを抜き出し，それらの予約の対象となっているイベント名の集合を得る

```
{ x.eventName | x in set getAllTicketRecords()
                & x.userID = userid }
```

集合における限量式

- 限量式の例
  - forall x in set {1, 2, 3} & x + 3 < 7　　（ = true ）
    「すべてのxが条件を満たす？」
    全称限量子（∀）に相当
  - exists x in set {1, 2, 3} & x + 3 > 4　　（ = true ）
    「条件を満たすxが少なくとも1つある？」
    存在限量子（∃）に相当
  - exists1 x in set {1, 2, 3} & x + 3 > 4　　（ = false ）
    「条件を満たすxがちょうど1つだけある？」

インタプリタの基本的な利用法(２)

- スクリプト上にコマンド群を記述，まとめて実行
  - scriptコマンドに続けてファイル名
- ファイル検索先パスの設定
  - ファイル名を相対パスで指定する場合，パス設定に含まれるディレクトリを順々に探す
  - dirコマンドで表示(引数なし)，設定(引数でパスを指定)
  - 立ち上げ時はToolboxの実行ファイルがあるパス(Program Files，インストールフォルダ内のbin)のみが指定されている

スクリプトの例

- 配付資料内の 1-set-scripts.txt

```
print "Quantified Expressions"
print forall x in set {1,2,3} & x + 3 < 6
print exists x in set {1,2,3} & x + 3 < 6
print exists1 x in set {1,2,3} & x + 3 < 6
print forall x in set {1,2,3}, y in set {4,5,6} & x + y < 10
print forall x in set {{1,2,3}, {4,5,6}, {7,8}}
        & forall y in set x & 0 < y and y < 9
print forall x in set {{1,2,3}, {4,5,6}, {7,8}}
        & exists y in set x & y mod 3 = 1
```

改行はスライドの都合
（実際は入れない）

演習問題1－2

- 前スライドの個々の式について，評価結果を予測してみよ
- 実際にスクリプトを実行してみて結果を確認せよ
  - インタプリタ上で

    script C:¥...¥1-set-scripts.txt
  - 今回は絶対パスをエクスプローラ等からコピーして入力する
    - dir設定してみてもよい

集合における限量式

- 限量式の利用イメージ
  - どの学生も，同じ時間帯に行われる2つの講義に履修登録していることはない，という条件式

```
forall student in set getAllStudents()
    & forall class1, class2 in set
            getRegisteredClasses(student)
        & class1.name <> class2.name
            => class1.time <> class2.time
```

let be式

- ある条件を満たす要素を集合から抜き出す
  - 例: let x in set {1, 2, 3} be st x < 3 in x * x
    - 集合 {1, 2, 3} の要素のうち，x < 3 を満たすものを x に割り当てる
      （この場合 x には 1 または 2 のどちらかが入る）
    - それを用いて in に続く式を評価
      （この場合式の値は 1 * 1 または 2 * 2，すなわち 1 または 4 となる）
  - 値の選ばれ方はインタプリタ依存
  - 該当する値がない場合実行エラー

演習問題1－3

- 数値の集合 s に対し，以下のlet be式により抜き出される値 x はどのような値か？

```
let x in set s be st
    forall y in set s & x >= y
        in …
```

- 具体的な値でやってみてもよい

```
let x in set {3, 6, 7, 9} be st
    forall y in set {3, 6, 7, 9} & x >= y
        in x
```

抽象的な記法の意義

- 早い段階での要求，仕様，設計の記述
  - 開発プロセス内のその時点で，何を検討し決める必要があるのか？何を決める必要がない（または決められない）のか？
- VDMの記述において捨象できるもの
  - 計算量やメモリ利用を考慮した実現方式
    - HashSet or TreeSet?
  - 効率を考慮した詳細なアルゴリズム
    - どうfor文でイテレーションを書く？効率をよくするためにはループの終了条件をどこでどう判定する？

抽象的な記法の意義

- 「何が必要で何が不要か」は，目的次第
  - とりあえず動かしてみてGUIと連結し，エンドユーザによる妥当性確認を行いたい
  - ➡ let be式などインタプリタ任せの実現でも，とりあえず動けばよい
  - 実装チームに渡す基本設計を構築，検証したい
  - ➡ 具体的なデータ型の選択やアルゴリズムの選択は詳細に定めず実装チームに委ねるが，システムにとって本質的，かつ複雑なアルゴリズムだけはfor文やwhile文を用いて記述，検証しておく

目次

- VDM概要（手法・ツール）
- 抽象的・宣言的な構文の活用
- 全体の形式仕様記述
- 一般論・補足

例題

- イベント登録管理システムの簡易版
  - 特定のイベントに対し，1名ごとの予約登録を管理
    - 「該当イベントにユーザ1が登録」
    - 「該当イベントにユーザ1～100から1人抽選で選んで登録」
    - 「該当イベントへのユーザ1の登録をキャンセル」
  - 該当イベントには固定の定員がある
    - 「該当イベントには最大30名しか登録できない」

VDM++記述の構造

class *className*（is subclass of *superClassName*）
types …　　-- 型定義ブロック
instance variables …　　-- インスタンス変数定義ブロック
values …　　-- 値定義ブロック
functions …　　-- 関数定義ブロック
operations …　　-- 操作定義ブロック
sync …　　-- 同期制約定義ブロック
thread …　　-- スレッド定義ブロック
end *className*

これらのブロックを
必要に応じ記述
（順不同，同じブロックが
何度現れてもよい）

最初の例題では
型定義，状態定義，操作定義
のみ扱う

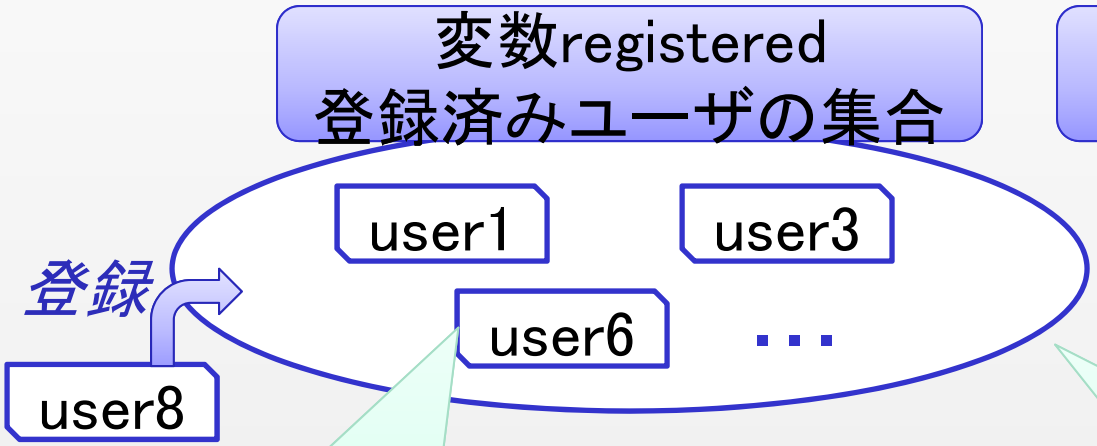
例題：VDM++記述の全体構造

```
class EventManager

types -- 型定義ブロック
… -- ユーザを表す型を定義

instance variables - インスタンス変数定義ブロック
… - 変数として「登録済みのユーザ集合」と「定員数」を持つ

operations -- 操作定義ブロック
-- register: 指定されたユーザをイベントに対し参加登録する
-- choose: 指定されたユーザの集合から，まだ参加登録していない
--                １人のユーザを選択し，イベントに対し参加登録する

end EventManager
```

class宣言とend宣言で囲む

VDM++ では -- （ハイフン2つ）から行末までがコメント扱い

例題：システム状態の抽象モデル化



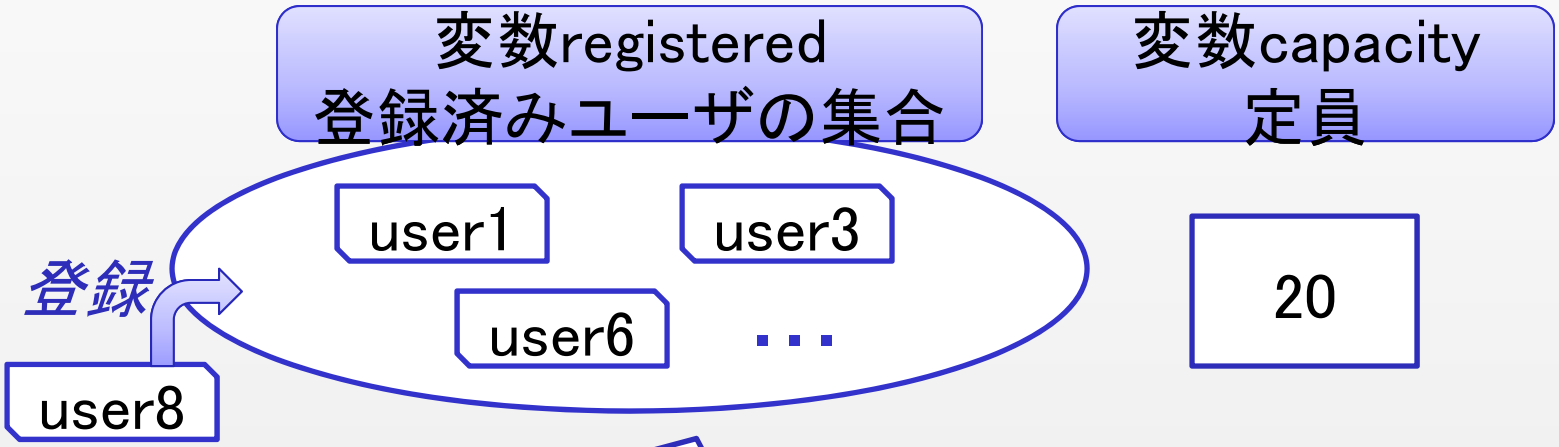
変数capacity
定員

20

ユーザ情報の型を具体的に決めず，
「何か個々を識別できる値」
と見なしてしまう

実装効率等は考えず，
集合として「情報を保持」
ということだけ表現

例題：システム状態の「正しさ」



不変条件

システムの状態が常に満たすべき条件

registered内のユーザ数 ≦ capacity

（すると登録操作を受け付けてもよい前提条件は？）

例題：型定義

```
types -- 型定義ブロック

public UserID = token; -- ユーザIDを表す型
```

組み込みの型や定義済みの型を用いて，新たな型を定義

ブロック内の一つ一つの定義はセミコロンで終える

トークン型は，「同じかどうか」の比較しかしない（識別子として用いる）値を表す

トークン型の値は，mk_token("user1") や mk_token(3) のように，任意の値を mk_token( ) で囲んだものとなる

実際にはユーザ情報はID，名前，住所といった情報を含むレコード構造を持つかもしれないが，現時点では不要な詳細として捨象している

例題：インスタンス変数定義

```
instance variables -- インスタンス変数定義ブロック

private registered : set of UserID := {}; -- 参加登録済みのユーザの集合
private capacity : nat; -- イベントの最大定員
inv card registered <= capacity; -- 不変条件
```

natは自然数型
（0以上の整数，natural）

invから始まる行は
不変条件（invariant）

- アクセス制御子は public，protected，private
（省略時はprivate）
- 「インスタンス変数定義ブロック」ではあるが，
staticキーワードによりクラス変数の定義も可能

例題：コンストラクタ定義

```
operations -- 操作定義ブロック

-- コンストラクタ
public EventManager : nat ==> EventManager
EventManager(cap) ==
  capacity := cap;
```

シグネチャ定義：
アクセス修飾子，操作名，
引数の型，戻り値の型

操作の本体は代入文
（定員の値を設定）

コンストラクタの戻り値の型は，このクラス自身
コンストラクタでは return 文は省略可能

例題：簡単な操作定義

```
operations -- 操作定義ブロック

-- 登録されたユーザー覧を得る
public getRegisteredUsers : () ==> set of UserID
getRegisteredUsers() ==
  return registered;
```

() は戻り値がないことを表している
(void)

操作の本体はreturn文

例題：登録操作定義(１)

- ユーザの参加登録を行う操作registerを定義
  - 引数として受け取ったUserIDを集合registeredに追加する(戻り値なし)
  - 事前条件：実行前に成り立っていなければならない前提条件
    - 登録済みユーザは定員に達していない
      (不変条件を崩すような実行をしないようにする)
    - 引数のユーザは登録済みではない
      (混乱を招きそうな，意味のない実行は行わせないものとする)

例題：登録操作定義(１)

```
operations -- 操作定義ブロック

-- 指定されたユーザを
-- イベントに対し参加登録する
public register : UserID ==> ()
register(user) ==
  registered := registered union {user}
pre
  card registered < capacity
    and
  user not in set registered ;
```

操作の本体は代入文：
受け取った引数を { } で囲むことにより
それのみを要素として含む集合を作り，
registered との和集合を求め(union)，
registered の値を更新する

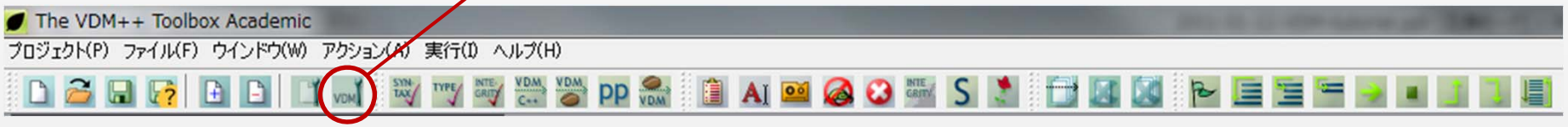
前述の2つの事前条件(pre はpreconditionの意)

セミコロン(;)はこの操作定義の最後にしか用いていない点に注意

Toolboxのセットアップ

- メニュー「プロジェクト」→「ツールオプション」

ツールバーだとこのボタン

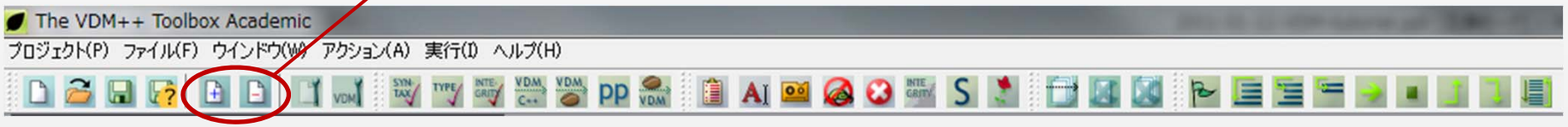


- 好きな外部エディタを選択
- 「フォント」タブ
  - 好きなフォントを選択
  - 文字コードはShift-JIS（日本語を扱う際に必須）

ファイル管理

- メニュー「プロジェクト」→「ファイルを追加・削除」

ツールバーだとこれらのボタン

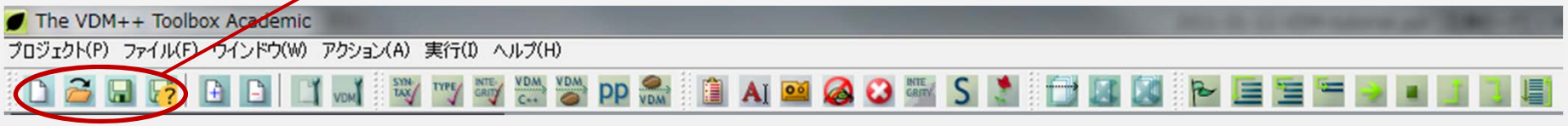


- 読み込みたいファイルを(複数)選択して追加
  - 今回は 2-EventManager.vpp および,
    2-EventManagerTest.vpp を読む
  - 読み込むと自動的に構文チェックが行われる
- または外したいファイルを選択して削除
  - Toolboxが読み込まなくなるだけ

プロジェクト管理

- メニュー「プロジェクト」→
  「新規，開く，保存，別名で保存」

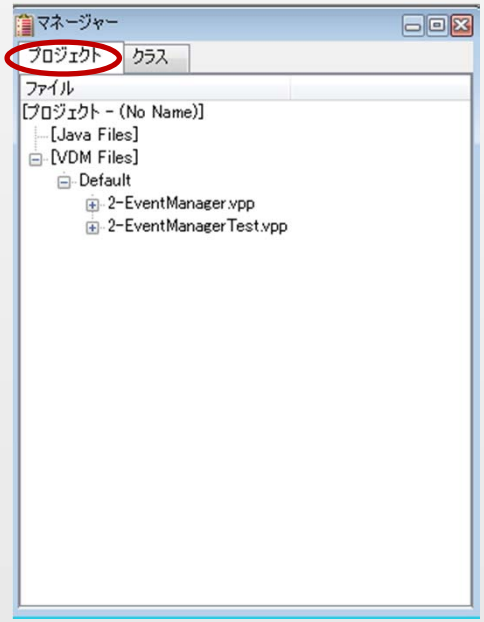
ツールバーだとこれらのボタン

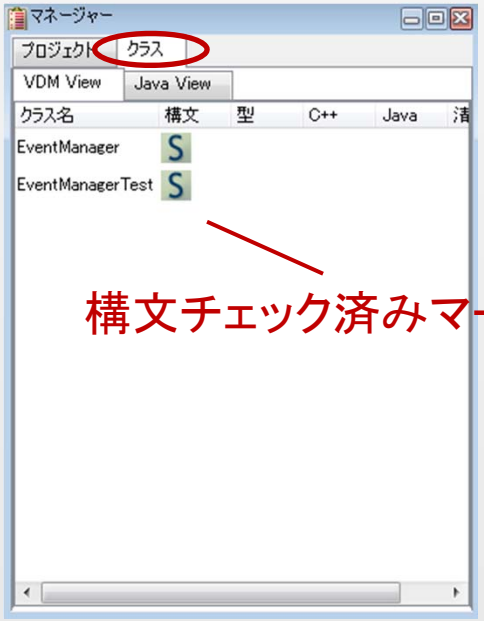


- 開いたファイル群（および関連設定）をプロジェクトとしてファイルに保存・読み出しできる（拡張子prj）
  - インタプリタのファイル検索先パスに追加される（dirコマンドで参照・設定するもの）

各クラスの状態

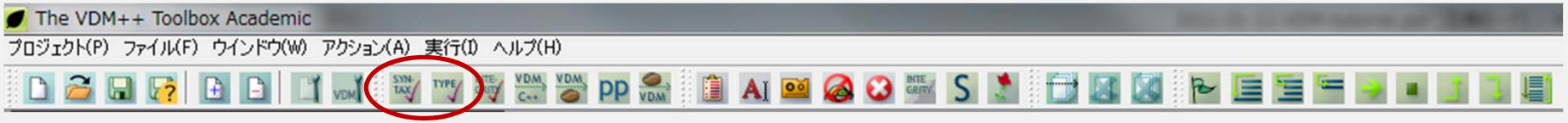
- マネージャーウィンドウが見えているはず
  - ない場合，メニュー「ウィンドウ」→「マネージャー」
    - （または該当ツールバーボタン）





クラスの操作

- マネージャーウィンドウで該当vppファイル（またはクラス）を選んで，メニュー「アクション」またはツールバー上のボタンから諸々操作



まずは構文チェック，型チェック
（その他Javaコード生成，清書，などなど）

- ファイルまたはクラスをダブルクリックでToolbox内のビューアが開く

VDM++ Toolbox利用の流れ

- 基本的な流れ
  - 外部エディタでvppファイル記述 → 読み込む
  - 構文チェック（自動で行われる），型チェック，インタプリタで実行
  - 問題があれば修正，反復
    - 外部エディタでの変更は反映されるはず
    - されないときはファイルをプロジェクトから削除・再追加してみる
  - プロジェクトとして保存
  - 読みだして，作業再開・・・

インタプリタ実行

- 1ユーザを登録して，登録情報を出してみる

```
init
create man := new EventManager(3)
print man.register(mk_token("u1"))
print man.getRegisteredUsers()
```

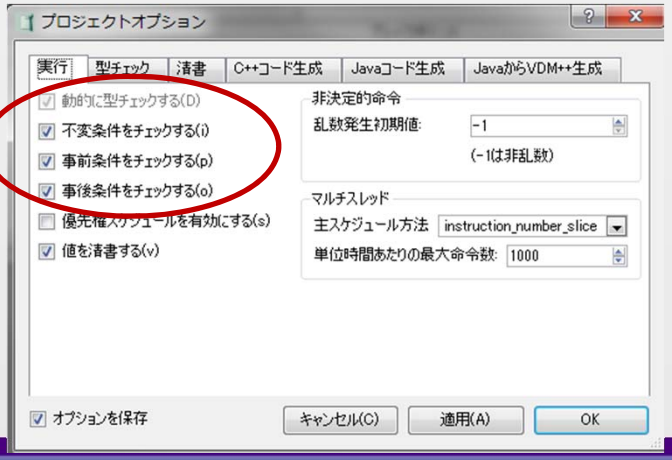
- インタプリタは状態を覚えている
  - initコマンドにより初期化する
    （VDM++記述が変更された場合には，明示しなくてもprintコマンド等の実行前に初期化が行われる）

検証のための実行設定

- メニュー「プロジェクト」→「プロジェクトオプション」

ツールバーだとこのボタン

- 「動的に型チェック」,「不変条件・事前条件・事後条件をチェック」が有効になっていることを確認



テスト実行

- 簡単なテストメソッドを別クラスにて定義，実行

```
class EventManagerTest

types
public UserID =  EventManager`UserID;

instance variables
manager : EventManager := new EventManager(3);

values
u1 : UserID = mk_token("User1");
u2 : UserID = mk_token("User2");
…
```

クラスEventManagerにて定義した
UserIDへのエイリアスを作っておく

テストメソッドにて用いる
ユーザを定数として定義しておく

テスト例

```
public test1a : () ==> set of UserID
test1a() == (
  manager.register(u1);
  manager.register(u2);
  return manager.getRegisteredUsers()
);
```

問題ない実行の例

```
print new EventManagerTest().teat1a()
```

テスト例

```
public test1b : () ==> set of UserID
test1b() == (
  manager.register(u1);
  manager.register(u2);
  manager.register(u3);
  manager.register(u4);
  return manager.getRegisteredUsers()
);
```

定員を超える登録をしようとするので
事前条件違反が検出される

もしも該当する事前条件を消しておくと，
登録が行われた時点で（registeredの値が更新された時点で）
不変条件違反が検出される
（事前条件により，不変条件が成り立たない状態に
つながるような操作呼び出しが防止されている）

文法メモ：セミコロンを付ける場所

- 定義ブロック内の，各定義の終わりにはセミコロンを付ける
- 複数の文を順次実行する際には，丸括弧で囲んでブロック文を書く

```
(
    s1;
    s2;
    s3
)
```

  - セミコロンは次の文との区切りに使い，ブロック文内の最後の文（上で s3 の後ろ）には不要
（VDM++ Toolboxではつけてもエラーにならない）

文法メモ：セミコロンを付ける場所

```
public op1 : () ==> int
op1() ==
  return x;
```

op1の定義を閉じる（必要）

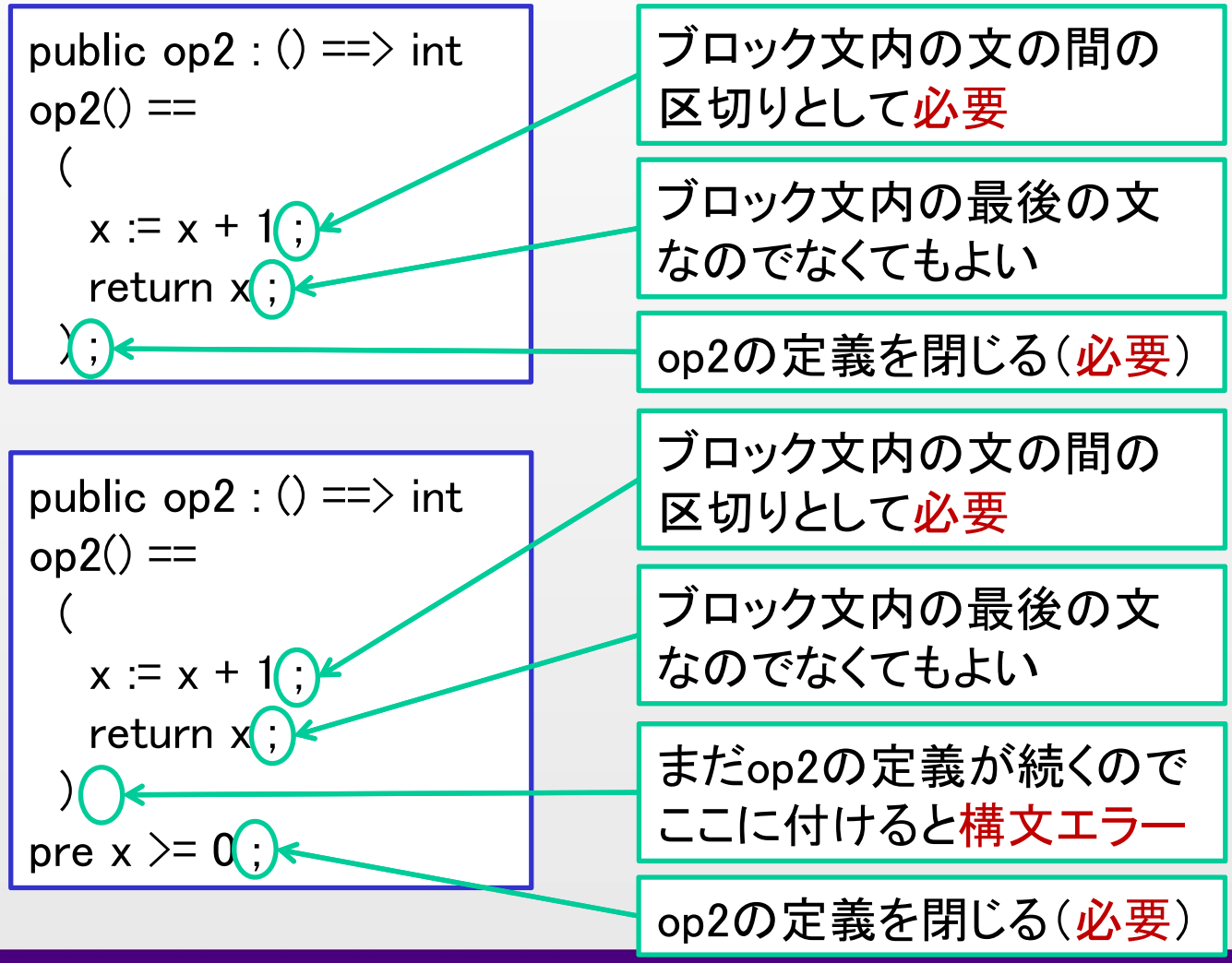
```
public op1 : () ==> int
op1() ==
  (
    return x;
  );
```

ブロック文内の最後の文なのでなくてもよい

op1の定義を閉じる（必要）

文が1個なら中カッコで囲まなくてもよい

文法メモ：セミコロンを付ける場所



演習問題2－1

- 以下の操作定義を追加せよ
  （事前条件は各自検討せよ）
  - ユーザの参加登録を取り消す操作unregister
    - 引数として受け取ったUserIDを集合registeredから取り除く（戻り値なし）
  - 複数のユーザを参加登録する操作registerAll
    - 引数として受け取ったset of UserIDを集合registeredに加える（戻り値なし）
- 構文チェック・型チェックを行い，適当なテストを実行してみよ

例題：登録操作定義(２)

- あるユーザを選んで参加登録を行う操作chooseを定義
  - 引数として受け取ったUserIDの集合のうち1つを集合registeredに追加し，そのUserIDを返す
  - 事前条件
    - 登録済みユーザは定員に達していない
      （不変条件を崩すような実行をしないようにする）
    - 引数のユーザ集合の中に，少なくとも1人は登録済みでないユーザがいる
      （この操作の機能が実現できない状況を除く）

例題：登録操作定義(２)

- あるユーザを選んで参加登録を行う操作choose を定義(続)
  - 記述の方針1：事後条件のみ定めることにより，この操作により起きる変化に対する要件のみを与える
    - 引数の集合内のうち，まだ登録されていないあるユーザが登録された状態に変化する
  - 記述の方針2：実行もできるように本体記述も与えておく
    - 今回はlet beを用いて簡易に実現

例題：登録操作定義(２)

```
-- 指定されたユーザの集合から，まだ参加登録していない
-- １人のユーザを選択し，イベントに対し参加登録する
-- 戻り値はこの操作で登録されたユーザ
-- 陰定義版
public choose : set of UserID ==> UserID
choose(users) == is not yet specified
pre card registered < capacity
    and exists user in set users & user not in set registered
post registered = registered~ union {RESULT}
     and RESULT in set users
     and RESULT not in set registered~;
```

操作本体はまだ記述しない

前述の2つの事前条件

事後条件（postは
postconditionの意）
詳細は次スライド

例題：登録操作定義（２）

```
post registered = registered~ union {RESULT}
    and RESULT in set users
    and RESULT not in set registered~
```

事後条件

戻り値となるユーザは
- 実行後には追加登録されている（そのユーザだけが増えている）
- 引数に含まれている
- 実行前には登録されていない

事後条件では，
入力および操作の実行前の状態と，
出力および操作の実行後の状態
の関係を書くために，以下の記法を利用する

- 操作の実行前の状態変数の値を参照するために ~ をつける
- 戻り値を指すキーワード RESULT を用いる

例題：登録操作定義（２）

```
public chooseImpl : set of UserID ==> UserID
chooseImpl(users) ==
  let x in set users be st x not in set registered
    in (
      register(x);
      return x
    )
pre
  …
post
  … ;
```

let be文による実行可能な本体記述

テスト例

```
public test2a : () ==> UserID
test2a() == return manager.chooseImpl({u2,u3});

public test2b : () ==> UserID
test2b() == (
  manager.register(u3);
  return manager.chooseImpl({u2,u3})
);

public test2c : () ==> UserID
test2c() == (
  manager.register(u2);
  manager.register(u3);
  return manager.chooseImpl({u2,u3})
);
```

補足

test2c

```
public test2c : () ==> UserID
test2c() == (
  manager.register(u2);
  manager.register(u3);
  return manager.chooseImpl({u2,u3})
);
```

引数に，まだ登録されていないユーザがいないので，事前条件違反が検出される

もしも該当する事前条件を消しておくと，
let be文を実行しようとするが条件を満たす値がないので
実行時エラーとなる
（事前条件により実行時エラーを引き起こす
操作呼び出しが防止されている）

演習問題2－2

- 複数のユーザを選んで参加登録を行う操作multiChooseを定義
  - 引数は2つ
    - UserIDの集合：この引数の中から登録するユーザを選ぶ
    - nat型：この引数で指定された数だけ登録する
  - 戻り値：登録されたUserIDの集合
  - 次ページのシグネチャに対して，事前条件・事後条件を検討し，記述せよ
    - 本体は is not specified yet でよい

演習問題2－2

```
public multiChoose : set of UserID * nat ==> set of UserID
multiChoose(users, num) == is not yet specified
pre
  ?
post
  ? ;
```

参考：multiChooseの本体定義例(１)

```
public multiChooseImpl1 : set of UserID * nat ==> set of UserID
multiChooseImpl1(users, num) ==
  let s in set power users be st
        forall u in set s & u not in set registered
          and
        card s = num
      in
    (
      registerAll(s);
      return s
    )
pre ...
post ... ;
```

let be文により，登録対象となるユーザ集合を選択させる
- その中のユーザは誰も登録済みではない
- 集合の大きさは引数で指定された数

登録して，返す

参考：multiChooseの本体定義例（２）

```
multiChooseImpl2(users, num) ==
  (
    dcl ret : set of User := {};
    for all user in set users do
      if user not in set registered then
        (
          ret := ret union {user};
          if card ret = num then
            (
              registered := registered union ret;
              return ret
            )
        );
    exit <UnExpectedResult>
  )
```

dclでは一時変数を定義

for all文では集合の要素を
1つずつ取り出しながら処理

登録されていないユーザを
一時変数に1つずつ入れていき，
指定数に達したら止める

事前条件が満たされていれば
到達しないはずの箇所

到達したら例外を投げる

目次

- VDM概要（手法・ツール）
- 抽象的・宣言的な構文の活用
- 全体の形式仕様記述
- 一般論・補足

話していないこと：文法

- シグネチャ・事前・事後条件のみの関数・操作定義を陰定義，本体を含めた定義を陽定義という
- メソッド定義としては，関数（変数の読み書きなし）と操作（変数の読み書きあり）を区別する
  - 様々な副作用分析に備えた区別をしている
  - 関数は関数型言語と同様の記法を用いる

```
fun_euclid : nat1 * nat1 -> nat1
fun_euclid(a, b) ==
  let r = a mod b in
    if r = 0 then b
    else fun_euclid(b, r);
```

関数：入力のみから出力を計算

```
op_euclid : nat1 * nat1 ==> nat1
op_euclid(a, b) ==
  (
    dcl r := a mod b;
    if r = 0 then return b
    else return op_euclid(b, r)
  );
```

操作：変数を読み書き

話していないこと：文法

- 型に不変条件を付けることも
  - 例：natだけども255以下に限定

```
UserID = nat
inv id == id <= 255;
```

  - 例：サッカーチームの情報を表すレコード型において，勝ち点と勝ち数・引き分け数の関係

```
FootballTeam ::
  win : nat
  draw : nat
  lose : nat
  point : nat
inv team == team.point = 3 * team.win + team.draw
```

- マルチスレッド制御も宣言的に記述可能
  - 例：「readとwriteは同時に実行できない」

話していないこと：ツール

- IOなど標準ライブラリ
- VDMUnit
- カバレッジ計測
  - 実行していない部分の強調表示も可能
- 証明課題生成
  - 潜在的なエラーの原因を列挙（完全ではない）
- UMLリンク
  - XMI形式のクラス図と相互変換
- コード生成
- 外部プログラム（GUIやテストツール）との接続

話していないこと: ツール

- Overture IDEにおける正規表現を用いたテストの網羅的な生成

```
values
testUsers : set of EventManager`UserID =
  { mk_token("User1"), mk_token("User2"),
    mk_token("User3"), mk_token("User4")};
instance variables
testman : EventManager := new EventManager(3);
traces
Test1:
  (let u in set testUsers in testman.register(u)
    |
   let u in set testUsers in testman.unregister(u))
   {1,4}
```

「4ユーザの誰かを選んで
登録または登録キャンセル」
ということを1回～4回繰り返す

8 + 8^2 + 8^3 + 8^4
= 4680通りのテストケースが
実行される

VDMのまとめ

- 状態（変数）と機能（関数・操作）を抽象的にモデル化，厳密に記述する
- 状態における不変条件，および関数・操作の事前・事後条件を明示化する
- 制約条件をチェックしながらインタプリタ実行し，検証・妥当性確認を行う
- VDMを用いる目的，そのためのモデル化や検証方針を明示的に意識・検討することが重要である