Java で Scheme を書いてみれば

湯浅太一[†]

Scheme 処理系を Java を記述言語として開発した．その経験をもとに，Lisp 系言語の処理系を Java で開発する際の長所・短所を検討する．

1. はじめに

今回開発した Scheme[1, 2] 処理系は，Java[3] の提供する豊富なクラスライブラリを，Scheme のもつ対話的な環境で利用することを最大の目的としている．クラスライブラリには，ネットワーク通信，スレッド，アプレットなど今後のソフトウェアに不可欠な要素を多く含んでいる．これらを Scheme から利用できれば，近代的なソフトウェアを開発するための，効率の良いプログラミング環境が容易に構築できるはずである．

この処理系はぶぶと呼び，この目的を達成するために Java で記述されている．筆者はけっして Java に精通しているわけではない．むしろ，Scheme 処理系を Java で開発することによって，Java を学習した．その過程で，他の言語（特に C 言語）で開発する場合との相違が明確になった．本稿は，この経験に基づいて，Java で Lisp 系言語処理系，あるいは他の言語処理系を開発する際の，長所と短所を検討するものである．

本稿はぶぶの実装を報告するものではないので，実装の詳細は避け，Java の検討に必要な部分だけをとりあげる．具体的な Java コードの例もあげるが，説明を簡単にするために，実際の処理系コードではなく，簡略化したものである．例えば，説明のために必要でない `public` 指定は省略したり，説明なしでも用途が想像できるような変数名を使ったりしている．

[†]京都大学大学院情報学研究科

2. スタック

一般に Lisp 系言語処理系を C 言語や Java で実装するためには，記述言語の使用する実行時スタックとは別に，処理系が利用するスタックが必要になる．Lisp 系言語では，可変個の引数を受け取る関数が定義できたり，引数として与えられたオブジェクトを 1 本のリストにまとめたり，逆にリストの個々の要素を引数として関数に与えたりといった機能がある．これらのために，スタック上に積まれた引数を処理系が直接操作する必要があるためである．さらに Scheme 処理系では，first-class の継続を実現するために，スタックの内容をヒープに退避したり，退避した内容をスタックに書き戻して実行する必要がある．また，末尾再帰呼出しの最適化を行うためには，実行中の関数フレームを書き換えることによって，新しい関数フレームを積むことなく関数を呼び出す処理が必要である．

処理系が利用するスタックは，1 次元の配列として表現するのは当然だが，Java で記述する場合は難しい．スタックには，パラメータや局所変数などを割当てるために，任意のオブジェクトを格納できなければならない．一方，関数フレームのリンク（動的リンクや静的リンク）を表現するために，スタックの場所を指すポインタも格納する必要がある．ところが，Java には，このようなポインタが存在しない．Java のオブジェクトは，構造体へのポインタとして実装されており，実装上はポインタが存在するが，配列の一部や変数などの場所を指すポインタは言語仕様上は存在

しないので，Java で Lisp 系処理系を記述する場合には使えない．しかし，スタックを配列として実現すれば，配列のインデックスでポインタを代用することが可能である．

ここで問題になるのが，インデックスは `int` 型であり，Java の int は primitive な型であり，オブジェクトを格納する配列 `Object[]` には格納できない点である．`Object[]` に格納するためには，`int` 型の整数値を，`Integer` クラスのオブジェクトに変換する必要がある．しかし，関数フレームを積むたびに，リンクを表現する Integer オブジェクトを生成するのはきわめて効率が悪い．そこでぶぶでは，スタックと同じサイズの Integer 配列（Integer プールと呼ぶ）を用意することによってこの問題を解決している．Integer プールの i 番目の要素には，整数 i を表現する Integer オブジェクトを起動時に格納しておく．スタックポインタをスタックに積むときには，スタックポインタを表すインデックス位置にある Integer プールの要素を積むことにしている．なお，この Integer プールは，数値計算の結果生成される int データをオブジェクトに変換する際にも使用する．

この方法で，スタック操作が Integer オブジェクトを生成する無駄は解消したが，スタックに積まれた Integer 型のスタックポインタを参照するときには，Java の言語仕様上，明示的なキャストが必要になる．例えば，フレームをポップするときには，

```
bp = ((Integer) stack[bp]).intValue();
```

といったコードが実行される．スタックに保存しておいた古いベースポインタを取り出して，それを新しいベースポインタとするコードである．保存されているベースポインタ `stack[bp]` は Integer オブジェクトなので，`Integer` クラスのメソッド `intValue` を使って int 値を取り出すのだが，スタックの要素は Integer オブジェクトとは限らないので，`intValue` を呼び出すためには，`Integer` へのキャストが必要である．C 言語であれば，この場合のキャストは，単にコンパイラの型チェックではねられないためのもので，実行時のオーバーヘッドはない．しかし Java では，実行時の型チェックを行うために，オーバーヘッドとなる．

3. オブジェクト

Java には豊富なクラスライブラリが用意されている．これらを有効に利用すれば，Lisp 系言語処理系が効率よく実現できる．

当然ながら，ぶぶがサポートする Scheme のすべてのデータ型は，Java のクラスとして定義した．このなかには，Java が提供するクラスをそのまま使用したものもある．さらにぶぶでは，後述のオブジェクト指向機能を使って，ユーザが任意の Java クラスを処理系にロードして利用することが可能である．これらの Java クラスのインスタンスが生成されれば，それらは Scheme のデータと同様に，変数に代入したり，関数への引数として使用できる．つまり，処理系実装用のものを含め，すべての Java クラスのインスタンスは，ぶぶでは first-class object になる．Java のクラス階層の最上位に位置する `Object` クラスは，ぶぶにおいてもクラス階層の最上位に位置する．

3.1 数値

Java の数値関係のクラス階層の最上位に位置するクラスは `Number` である．`Number` にはいくつかのサブクラスがあるが，現在のぶぶの実装では，そのうち，`Integer`，`Double`，`BigInteger` だけを使っている．

整数データは，32 ビットで表現できるものは Integer で，それ以外（いわゆる bignum）は BigInteger で実装している．Java の BigInteger は，任意長の整数を実装したクラスであり，Scheme の実装に必要な整数演算のすべてがメソッドとして定義されている．Lisp 系言語処理系を構築するために導入されたのではないかと思えるほどである．ただし，Java は BigInteger と Integer の間の自動変換をしてくれない．整数を値とする BigInteger のメソッドは，結果が Integer で表現可能であっても，BigInteger を値とする．このために，処理系が演算結果の BigInteger を正規化す

る必要がある．正規化しなくても，言語仕様上はいっこうに問題にならないが，数値関係の組込み関数で型ディスパッチをする際には，できるだけIntegerで表現しておくほうが効率がよい．

逆に，Integerの演算を行った結果がIntegerでは表現できない整数になることがある．このために，数値関係の組込み関数は，Integerの演算はint（32ビット符合付き整数）ではなく，long（64ビット符合付き整数）で行う．例えば，Integerの値xに1を加えるコードは，次のように記述される．

```
long y = ((long) x.intValue()) + 1;
if (y == (int) y)
  return makeInt((int) y);
else
  return BigInteger.valueOf(y);
```

bignumをサポートするLisp系言語の処理系開発においては，bignum演算のコーディングにかなりの時間を要する．今回BigIntegerを使えたことで，処理系の開発期間がおおはばに短縮できた．

処理系には，上記の3つ以外の数値クラスがロードされて使われる可能性がある．Javaが提供する数値クラスは他にもあるし，`Number`のサブクラスを，ユーザが定義してロードするかもしれない．数値関係の組込み関数が，これらのクラスに対しても対応できることが望ましい．幸いなことに，`Number`クラスには，`doubleValue`というインスタンスメソッドが定義されている．数値オブジェクトをdoubleに変換するものである．`Number`クラスはアブストラクトクラス（直接のインスタンスを生成できない）であり，そこで定義された`doubleValue`もアブストラクトメソッドである．このために，`Number`を継承するクラスは，（アブストラクトクラスでない限り）独自の`doubleValue`メソッドを定義する必要がある．つまり，任意の数値オブジェクトは，`doubleValue`を使えばdoubleに変換できる．この特性を活かして，もし上記3つ以外のクラスが数値関係の組込み関数に与えられた場合は，`double`に変換してから演算を行うことにした．これによって，多少の不都合が生じるかもしれないが，とにかく演算結果を得ることができる．例えば，ユーザが分数のクラスを`Number`のサブクラスとして定義しても，Schemeの数値として扱うことができる．ただし，分数どうしを足しても，結果は分数（あるいは整数）にはならず，浮動小数点数となる．

3.2 文字と文字列

文字データは，Javaの`Character`クラスで実装した．特定の文字コードを仮定したコーディングはいっさい行っていないので，Javaのランタイムシステムが日本語対応になっていれば，処理系も自動的に日本語対応になる．

セルの消費をおさえるために，同じ文字は，同じCharacterオブジェクトで表現したい．ASCII文字の127個は当然であるが，その他の，日本語文字なども同一文字同一オブジェクトにするのが望ましい．一度生成された文字オブジェクトをハッシュテーブルに記憶しておいて，文字コードから文字オブジェクトへの変換時に使用すれば実現可能である．Javaには`Hashtable`というクラスがある．当初はこれを使おうと試みたが，`Hashtable`のキーは，Objectでなければならない．文字コードはintなので，文字コードをキーとしたハッシュテーブルは，このクラスを使って実現できない．配列を使って処理系独自のハッシュテーブルを実装するしかない．Javaの多くのプログラマもそうであろうが，必要なクラスライブラリを探して，それらを組み合わせることでJavaプログラムを作成しようとする．みずからハッシュテーブルを作ろうという気にはなかなかならないものである．筆者も面倒になって，同一文字同一オブジェクトは，結局実現しなかった．ただしASCII文字だけは，処理系同時に配列にすべて生成しておいて，同一文字同一オブジェクトとなっている．

文字列データも，Javaの`String`クラスをそのまま使って実装した．ここで問題となったのが，JavaのStringがimmutableである点である．文字列中の文字を，別の文字で置き換えることができない．一

方，Schemeの文字列はmutableであり，文字を置き換える関数`string-set!`が仕様に含まれるが，この関数は実装できなかった．これが，Schemeの言語仕様で，ぶぶがサポートしていない唯一の機能となった．文字列データを，文字の配列として実装したり，文字列をオブジェクト変数に格納する新しいクラスとして実装すれば，この問題は解決する．しかし，それにともなう文字列操作のオーバーヘッドを考慮すれば，`string-set!`を未定義とするほうが，賢明であると判断した．

3.3 リスト，ペア，空リスト

ここでリストとは，ペア（コンスともいう）に空リスト（nil）を加えたものである．Javaにはこれらに対応するクラスが用意されていないので，独自に用意する必要がある．

リスト処理関数のコードには，リストを格納する変数がひんぱんに現れる．自然と，リストを実装するクラスが必要になる．名前は`List`とした．nilは`List`のインスタンスとし，`List`クラスに`NIL`という定数（正確には，static final 変数）を用意して，処理系起動時にnilを格納しておく．処理系のコード中で，`x`の値がnilかどうかを判定するには，

```
x == List.NIL
```

とする．ペアかどうかは，

```
(x instanceof List) && x != List.NIL
```

によって判定することも不可能ではないが，独自のクラス（`Pair`とした）を用意するほうが効率がよい．

```
x instanceof Pair
```

Lisp系言語は，ペアと同様に，nilに対してもcar部とcdr部を持たせていることが多い．どちらの値もnil自身である．ぶぶでもこの仕様にした．`List`クラスに`car`と`cdr`というインスタンス変数を定義し，処理系起動時に，nilのcar/cdr部をnilに設定する．`Pair`は`List`のサブクラスなので，これらのインスタンス変数を継承する．このために，`Pair`は単にペアをnilと区別するためだけのクラスになってしまった．その定義は，コンストラクタの定義

```
Pair (Object a, Object d) {
  car = a; cdr = d;
}
```

だけの簡単なものである．

このように定義したリストは，Javaの場合は効率が悪い．リストをたどる処理が，実行時の型チェックを伴うからである．例えば，リストの長さを求める組込み関数`length`のコードは次のようになる．ここで，`x`には`length`への引数が格納されているものとし，その型は`Object`である．

```
int n = 0;
while (x instanceof Pair) {
  x = ((Pair) x).cdr;
  n++;
}
if (x != List.NIL) error;
```

while文の条件で`x`の値がPairであることを確認しているので，cdrをとるときの型チェックは不要である．にもかかわらず，Javaのコンパイラは，実行時に型チェックを行うコードを生成する．もしこのチェックをはずしたコードを生成すると，ランタイムシステムのセキュリティチェックにかかってしまい，コードを実行することができない．チェックをはずす最適化を実現するのは容易であろうが，JVMコードを受け取るランタイムシステムが，JVMコードを解析してチェックなしでも安全であることを確認するのは困難だからであろう．

処理系が内部で使用するデータにも，リストとして表現されるものがある．それらのほとんどは，真のリスト（cdr部をたどるとnilで終るリスト）であることが保証されている．そのようなリストを`List/Pair`で表現するのは効率が悪いので，真のリストを表現するためのクラス`TrueList`を別途用意することにした．cdr部の型は，`Object`ではなく，`TrueList`である．TrueListの`x`の長さを求めるコードは，

```
int n = 0;
while (x != List.NIL) {
  x = x.cdr;
```

```
    n++;
  }
```

となり，実行時の型チェックが不要となる．

4. スレッド

Javaには大域変数がない．Lisp系言語処理系を作るときには，スタックのベースポインタや記号表といった大域的な情報が必要である．Javaで記述する場合は，これらの情報をクラス変数あるいはインスタンス変数に格納するしかない．

Javaにはスレッドがある．これを有効に利用すれば，スレッド機能を持つ処理系が容易に構築できる．大域的情報のうちで，ベースポインタなどのスタック関係の情報は，個々のスレッドに固有である．一方，記号表などのヒープに関連する情報は，全スレッドに共通である．このために，前者をスレッド・オブジェクトのインスタンス変数に格納し，後者はクラス変数（記号表なら，`Symbol` クラスの）に格納するのが自然である．

ぶぶは，first-classの継続をサポートするために，Schemeコードを独自の仮想機械のコード（過去の経緯から，バイトコードと呼んでいる）に変換し，仮想機械のバイトコード・インタープリタ（BCI）で実行する．BCIはバイトコード列を1命令ずつ解釈実行するループとして実現されている．このために，いったんBCIが動作を始めれば，Javaのスタックが伸びることがない．もちろん，処理系の内部的な処理のためにJavaのメソッド呼出しを行えばスタックは伸びるが，次のバイトコード命令を実行するときには，スタックはもとの状態に戻っている．捕捉した継続を実行するときには，捕捉時のJavaスタックの状態は不要であり，Javaのスタックに依存しない継続が実現されている．

BCIは，Schemeプログラムを実行する主体であり，個々のスレッドに一つずつ必要である．つまり，スレッドとBCIは1対1に対応する．そこで，BCIのクラスは，`Thread` のサブクラスとして定義した．ぶぶにおいて新しいBCIを生成することは，新しいスレッドを生成することに他ならない．当然，スレッドごとの大域的情報は，各BCIのインスタンス変数として実装している．これらの情報を参照する実装用のJavaメソッドはすべて，BCIオブジェクトを引数として受け取ることになる．

5. 組込み関数

Javaには関数がない．Schemeの組込み関数は，Javaのメソッドとして定義するしかない．

ぶぶの最初のバージョンでは，次の実装法を採用していた．

1. 組込み関数ごとにクラスを定義する．
2. 個々のクラスには，`exec` という名前のインスタンスメソッドを定義する．その本体で，組込み関数の動作を与える．`exec` はBCIオブジェクトを引数として受け取り，関数としての引数は，処理系のスタックから取り出す．
3. 処理系起動時に，すべてのクラスのインスタンスを一つずつ生成し，それを，関数名を表す記号にバインドする．このインスタンスが，関数オブジェクトとして使われる．
4. 組込み関数を呼び出すには，関数オブジェクトに対してメソッド `exec` を適用する．

例えば，組込み関数の `car` のためのクラス定義は，おおむね次のようになる．

```
class Car implements Function {
  void exec (BCI bci) {
    bci.acc = ((List) bci.arg[1]).car;
  }
}
```

ここで `Function` は，組込み関数用のクラス全体をまとめるinterfaceである．

```
interface Function {
  void exec (BCI bci);
}
```

この方法は確実に動作し，実行効率も悪くない．さらに，Javaのクラスローダを利用してautoloader機能が簡単に実装できるという利点がある．具体的には，

1. **Function** を implement する autoloader クラスを一つ用意する.
2. 処理系起動時に，個々の組込み関数の代用として，autoloader クラスのインスタンスを一つずつ生成する．その際，組込み関数用のクラス名と，バインドされる記号を，インスタンス変数に保持しておく.
3. autoloader クラスの **exec** メソッドは，組込み関数用クラスのインスタンスを一つ生成し，対応する記号にバインドしなおす．インスタンス生成時に，Java のクラスローダによって，自動的に組込み関数用クラスがロードされる.

この autoloader 機能を使えば，クラスのロードはオンデマンドで行える．関数ごとにクラスを定義する方法なので，クラスの個数が膨大になるが，処理系起動時には，autoloader クラスだけがロードされていればよい.

Java は基本的に，クラスごとにソースファイルを用意しなければならない．上の方法では，組込み関数ごとにソースファイルが一つずつ必要になる．しかも，そのほとんどは，上の **Car** の例でわかるように，きわめて小さな（数行程度の）ファイルである．これは，二つの大きな問題を生じることになった．一つは，処理系開発の手間が膨大になったことである．いくつかの組込み関数に共通した修正を行うには，対応するファイルすべてを変更する必要がある．**car** と **cdr** を組み合わせた関数用だけでも，**Cadr**，**Caddr**，**Cddddr** など 30 個のクラスができる．これらのコードが一つのファイルに格納されていれば，1 回の replace で簡単に終る修正でも，30 個のファイルを変更しなければならない．変更した後は，30 個のファイルをコンパイルしなければならない．これでは，効率的な処理系開発は望めない.

もう一つの問題は，Java のクラスファイルの構造に起因する．個々のクラスファイルには，ローダが必要とする情報が蓄えられている．比較的大きなクラスのクラスファイルではその情報量は問題にならないが，小さなクラスファイルの場合は，本来のクラス定義と比較して，はるかに大きな情報が蓄えられている．このために，処理系全体のクラスファイルの総量は膨大になってしまった．手元の計算機でローカルにぶぶを使う場合は気にならないが，ネットワーク経由でクラスファイルをロードすると，通信時間が問題となってくる.

Java には，reflection 機能がある．メソッドをオブジェクトとして取り出して，引数を与えて呼び出すことができる．現在の組込み関数の実装は，reflection 機能を使うことによって，上記の問題を解決している．具体的には，

1. 組込み関数を，適当なクラスの static メソッド（クラスメソッド）として定義する．互いに関連する組込み関数群のメソッドは，一つのクラスに定義する.
2. 組込み関数を表現するためのクラス **Function** を一つ用意する.
3. 処理系起動時に，組込み関数ごとに **Function** のインスタンスを一つ生成し，対応するメソッドを reflection を使って取り出し，インスタンス変数に格納する．対応する記号に，この関数オブジェクトをバインドする.
4. 組込み関数を呼び出すときは，関数オブジェクトに格納されたメソッドに対して，reflection 機能が提供する **invoke** メソッドを適用する.

例えば，リスト処理関係の組込み関数は，一つのクラス（実装では **List** クラス）に次のように定義する.

```
class List {
  ...
  static void car (BCI bci) {
    bci.acc = ((List) bci.arg[1]).car;
  }
  ...
}
```

メソッドを呼び出すための **invoke** は 2 引数である．呼び出すメソッドが static の場合は，第 1 引数は **null** でよい．第 2 引数は，呼び出すメソッドに受け渡す引数すべてを格納した配列である．一般的にはこの

配列は動的に生成するので，`invoke` によるメソッド呼出しにはオーバーヘッドが伴う．ぶぶの実装で使用する場合は，引数は現在実行中の BCI だけである．そこで，個々の BCI 生成時に，その BCI 自身を唯一の要素とする配列を用意しておき，それを `invoke` の第2引数として渡すことによって，呼出しごとの配列生成を回避している．関数オブジェクト `f` の呼出しコードは次のようになる．

```
f.method.invoke(null, myself);
```

以上の方法によって，当初の実装に伴った問題は解決する．この方法でも，autoloader 機能は簡単に実現できるが，クラスファイルがコンパクトになったためか，処理系起動時にすべての組込み関数を初期化しても，autoloader を初期化する時間とあまり差異が生じなくなった．このために，現在のバージョンでは autoloader の必要性がなくなってしまった．

6. GC

Java には GC（ごみ集め）がある．Lisp 系処理系の実装では，GC のバグを除去するために多大の時間と労力を要するのが常である．GC 以外のバグだと思って処理系の動作を追跡すると，実は GC のバグだった，ということも少なくない．GC のバグは，再現性が乏しい．デバッグするために式を入力すると，それによってヒープの状態が変化して，以前の現象が現れなくなることもある．しかし，GC を備えた Java で記述することによって，この悩みから解放される．

特に重要なことは，記述言語の提供する GC は，記述言語のスタックも GC のルートとして扱ってくれる点である．GC を持たない C 言語で実装した場合は，処理系の一部として GC を実装しなければならない．その場合，C 言語のスタックを GC のルートとして扱うためには，スタックの構造に依存するコードとなり，処理系の移植性が低下する．高い移植性を保つためには，C 言語のスタックからしかたどれない可能性のあるオブジェクトを，処理系のスタックに積むなどして，GC が誤って回収するのを防ぐ必要がある．これは実行時のオーバーヘッドとなる上に，スタックへの積み忘れがあれば，デバッグの困難な GC 関連のバグとなる．

処理系独自に開発した GC と比較すると，記述言語が提供する GC は，処理系のスタック構造を考慮できないなどの理由から，実行効率が劣るものと想像できる．この点に関しては，Scheme の末尾再帰呼出しの最適化機能は都合がよい．不要な関数フレームがスタックに残らないので，スタックが短く抑えられ，GC のルート・スキャン時間が短縮される上に，不要になったデータの回収時期が早くなるからである．

7. Java とのインターフェイス

ぶぶにはオブジェクト指向機能が実装されている．詳細は別の機会にゆずることにして，定義済みのクラスを操作する構文（実際はマクロ）のいくつかを次に示す．

(**new** *class-name*)
: クラスのインスタンスを生成する．

(**send** *object method-name* . *args*)
: *object* に対して，メソッドを適用する．

(**slot-ref** *object slot-name*)
: スロット（クラス変数またはインスタンス変数）の値を返す．

(**slot-set!** *object slot-name value*)
: スロットの値を変更する．

これらのマクロは，ぶぶのオブジェクト指向機能を使って定義したクラスに対して適用できるのはもちろん，Java で記述したクラスに対しても適用できる．例えば `new` に対して Java のクラス名を指定すれば，Java のクラスのインスタンスが生成できる．指定したクラスがまだ処理系にロードされていなければ，ロードしてからインスタンスを生成する．

ぶぶのオブジェクト指向機能は，Java のクラスライブラリを Scheme 処理系に取り込んで利用するのが本来の目的であったが，処理系のデバッグにも活用できることが分かった．処理系自体が Java のクラス集合として実装されており，これらの実装用クラスは，Java が提供するクラスライブラリと区別できない．こ

のために，実装用クラスに対してもぶぶのオブジェクト指向機能が利用できることになってしまった．例えば，Scheme レベルで通常は

```
> (define x (cons 1 2))
x
> (car x)
1
> (set-car! x 3)
3
> x
(3 . 2)
```

とするところを，オブジェクト指向機能を直接使って，

```
> (define x (new Pair 1 2))
x
> (slot-ref x car)
1
> (slot-set! x car 3)
3
> x
(3 . 2)
```

とすることもできる．この方法を使えば，特別なメカニズムを用意しなくても，Scheme レベルでは得ることのできない処理系内部のさまざまな情報を得ることができる．

実行中のスレッド（BCI オブジェクト）は，オブジェクト指向機能を使っても得ることはできない．しかし，簡単な組込み関数

```
static void getBCI (BCI bci) {
  bci.acc = bci;
}
```

を定義すれば，取り出せる．BCI オブジェクトに対してオブジェクト指向機能を使えば，実行状態，例えばスタックに関する情報を得ることができる．

```
> (define x (getBCI))
x
> (slot-ref x bp)
20
```

このように，オブジェクト指向機能を使えば，処理系の内部情報を簡単に取り出すことができ，処理系のデバッグにはおおいに役立った．しかしこの特性は，両刃の剣である．同様の操作が，一般のユーザにも利用できるからである．内部情報を取り出すだけであれば問題ないが，内部情報を変更すると処理系を破壊する危険性がある．例えば，

```
(slot-set! () car 1)
```

とすると，空リストの car の値が空リスト自身ではなく，1 になってしまう．

```
> (car ())
()
> (slot-set! () car 1)
1
> (car ())
1
```

この例の場合は，List クラスの car/cdr スロットを private と宣言すれば，「不正アクセス」は拒否することができる．スロットを直接参照できないようにし，これらのスロットを参照するためのメソッドを使うように改めればよい．car を取り出すメソッドは List クラスで定義し，car の値を変更するメソッドは Pair クラスで定義すれば，Pair のインスタンスではない空リストの car の値を変更することはできなくなる．しかし，car/cdr の参照は，実行時に頻繁に起きるので，処理系のコードがいちいちメソッドを呼び出していては，実行効率が大幅に低下する．現状では，これらのスロットはデバッグしやすいように public としているが，将来的には，Java のパッケージ機能を有効に利用するなどして，処理系の安全性を高める必要があろう．

8. 例外処理

Java には例外処理機能がある．一般に，Lisp 系言語処理系を Java で記述すると，次の場合に例外が発生する．

- 処理系のコードが Scheme レベルのエラー（引数の個数や型が正しくないなど）を検出したとき

- Javaのランタイムシステムが実行時エラー（スタックオーバーフローなど）を検出したとき

さらにぶぶでは，Javaのクラスライブラリを利用できるので

- クラスライブラリがエラーを検出したとき

にも例外が発生する．

ぶぶには，Javaの例外処理機能を使った例外処理機能が実装されている．Schemeプログラムで例外を発生するには，次の関数を使う．

(**throw** *exception*)

throwへの引数は，**Throwable**クラス（またはそのサブクラス）のインスタンスである．このクラスは，例外として投げることのできるすべてのオブジェクトのクラスである．**Throwable**のサブクラスとして，**Error**（主としてランタイムシステムが検出する，実行継続が不可能なエラー）と**Exception**がある．**Exception**のサブクラスである**RuntimeException**は，ランタイムシステムが検出する算術演算エラーなどを含む．**throw**への引数は，これらのクラスやそのサブクラスのインスタンスとして生成する．

例外処理を行うためには，次の構文を使用する．

(**with-handler** ((*class-name*$_1$ *handler*$_1$)
...
(*class-name*$_n$ *handler*$_n$))
body)

*body*を実行中に，いずれかの*class-name*の例外が投げられると，その例外オブジェクトを引数として，対応する*handler*を呼び出す．

ぶぶのこのような例外処理機能は，Javaの例外処理機能を使って実装しており，**Throwable**の任意のサブクラスを，**with-handler**に指定できる．これによって，Schemeプログラムが投げた例外のみならず，Scheme処理系やJavaのランタイムシステムが投げた例外に対しても，その処理をSchemeレベルで記述することができる．

実装はきわめて簡単である．BCIがすべての例外を捕捉し，**with-handler**のフレームを処理系スタックのトップから順に検索する．捕捉した例外を処理するフレームがみつかれば，フレームに格納されているハンドラを単純に呼び出す．もしみつからなければ，デバッガ（ブレークループ）に入る．BCIは基本的に，バイトコード命令を一つずつ実行するループである．そのループを，Javaのtry-catch式で囲むことで，捕捉を行っている．例外を捕捉した時点では，Javaのスタックは巻き戻っているが，処理系のスタックは例外発生時のままである．したがって，処理系のスタックを検索することでハンドラを探すことができる．この実装方法は**with-handler**の関数フレーム内にハンドラを格納するものなので，first-classの継続と相性がよい．Schemeプログラムの中で，例外を投げる際に継続を捕捉しておけば，ハンドラの処理が終った後に明示的に継続を投げて，中断した実行を再開することもできる．

Javaでは，各メソッドが投げる例外（そのメソッドが呼び出す別のメソッドが投げる例外も含む）のクラスを，明示的に宣言する必要がある．メソッド定義の先頭にthrows節

throws *class-name*$_1$, ..., *class-name*$_n$

を書いておかないと，Javaコンパイラが受け付けてくれない．ただし，ErrorとRuntimeException およびそれらのサブクラスは宣言する必要がない．**RuntimeException**以外の**Exception**クラスはchecked exceptionと呼ばれ，必ず宣言しなければならない．

ぶぶの組込み関数には，checked-exceptionを投げるものがある．Javaのreflection機能を使ってこれらの関数を呼び出すための処理系のメソッドは，可能性のある例外クラスのすべてをthrows節に並べておくことになる．しかしこれは現実的でない．そこで，checked exceptionをラッピングする方法を採用した．組込み関数を呼び出すコードは，**invoke**の実行中に発生した例外をすべて捕捉する．unchecked exceptionならそのまま投げ直すが，checked exceptionであれば，処理系が用意した**CheckedException**クラスのインスタンスを生成し，捕捉した例外をそのスロットに格納して投げ直す．**CheckedException**は

RuntimeException のサブクラスとして定義されているので，throws 節は不要になる．BCI が Checked-Exception を捕捉したら，そのスロットに格納されている本来の例外オブジェクトを取り出して，それをハンドラに受け渡す．

ぶぶのオブジェクト指向機能を使えば，任意の Java メソッドを呼び出すことができる．これらのメソッドも checked exception を投げる可能性がある．そこで，send 構文を実装するための処理系メソッドも，checked exception をラッピングする上記の手法を採用している．

9. おわりに

ぶぶは，first-class の継続をサポートするために，Scheme プログラムを独自のバイトコードに変換し，Java で記述されたバイトコードインタープリタで実行している．バイトコードインタープリタ自身は，Java コンパイラが JVM コードに変換したものを，JVM のインタープリタが実行する．二重のインタープリタで Scheme プログラムを実行することになり，Java 自体の実行効率が良くないこともあって，C 言語などで記述した処理系と比較すると，実行性能はけっして良いものではない．しかし，本稿で触れたような効率向上の配慮を行うことによって，まずまずの性能を備えている．簡単なベンチマークテストの結果では，Scheme プログラムを Java コードに変換して実行する KAWA[4] よりも，実行性能に優れているようである．

Scheme プログラムをバイトコードに変換するコンパイラは，TUTScheme[5] という，C 言語で記述した Scheme 処理系用に以前開発したものを基本的にそのまま使用している．バイトコードの命令セットも同じである．TUTScheme と比較すると，実行性能は 1/10 程度である．しかし，Java のクラスライブラリを利用できるというメリットは，この性能比を相殺できるほどの価値がある．

既存のコンパイラを利用したり，Java の提供するクラスを有効に使って開発したので，開発期間は本来ならば，短くてすんだはずである．しかし，Java によるプログラミング経験が皆無の状態で開発を始めたので，実際にはかなりの開発期間を要した．このために，開発効率についての議論を本稿に含むことができなかったのは残念である．

最後に，処理系の起動について言及しておく．本文で「処理系の起動時に …」という言い方を何度か行ったが，現在のぶぶ処理系は，起動されると実装用のクラスをすべてロードして初期化し，その後にコンパイラをロードする．この起動時の初期化の時間は動作環境に依存するが，やや待たされる，という印象を受ける．この待ち時間を解消するためには，一度初期化した処理系のイメージをファイルに保存しておいて，次回からはそれを起動するのがよさそうだが，その方法が分からない．よい方法があれば，筆者に連絡したいただきたい．

参考文献

[1] IEEE Standard for the Scheme Programming Language, IEEE (1991).

[2] Clinger, W. C. and Rees, J., editors. : Revised^4 Report on the Algorithmic Language Scheme, MIT AI Memo 848b, MIT (1991).

[3] Gosling, J., Joy, B., and Steele, G.: The Java Language Specification, Addison-Wesley (1996).

[4] Bothner, P.: Kawa, the Java-based Scheme System, www.cygnus.com/ bothner/kawa.html.

[5] 小宮常康, 湯淺太一: Future ベースの並列 Scheme における継続の拡張, Vol. 35, No. 11, pp. 2382–2391 (1994).