Java アプリケーション組み込み用の Lisp ドライバ

湯 淺 太 一[†]

Java 言語によって開発するアプリケーションに組み込んで使用することを主要目的として設計した Lisp ドライバを紹介する．設計にあたって重視した点は，(1) Lisp 処理系の実装ノウハウを持たない Java プログラマにも機能の追加・削除・変更が容易に行えること，(2) コンパクトな実装であること，(3) 性能が極端に悪くないこと，等である．これらの条件を満たすために，Java の持つ機能を有効に利用し，大域的な制御情報を排除し，自然な Java コーディングを採用して，ドライバを開発した．このドライバは，高度な Lisp プログラム開発支援ツールを備えていないが，単独で Lisp 処理系として利用することも可能である．Lisp の言語機能としては，IEEE Scheme のほぼフルセットをサポートしている．処理系のソースコードはわずか約 3500 行，100K バイト程度である．実行性能はけっして良くないが，許容できる範囲に収まっている．

A Lisp Driver to be embedded in Java Applications

Taiichi Yuasa[†]

We present a Lisp driver which is designed to be used primarily as an embedded system in Java applications. The key design issues include: (1) it should be easy to extend, modify, and delete the functionality even for a Java programmer who is not familiar with Lisp implementation, (2) the driver itself should be compact enough, and (3) the performance should be comparable, though not excellent. In order to develop a driver that solves these issues, we highly made use of Java features, avoided global control mechanisms, and applied widely acceptable Java coding. Although the driver is not equipped with powerful tools to support Lisp programming, it can be used as a stand-alone Lisp processor. It supports the functionality of nearly the full-set of IEEE Scheme. The current implementation consists only of 3500 lines or 100 Kbytes of source code. The runtime performance is not excellent, but remains in an acceptable range.

1. はじめに

ソフトウェア製品に Lisp 処理系を内部的に組み込むことは従来から行われている．Emacs エディタが，Emacs Lisp を内部に備えている[15] ことはよく知られているし，CAD システムのなかには，Lisp 処理系を組み込んでいるものがある[12]．コンパイラの開発にあたっても，中間言語やマシン記述が Lisp の S 式に似た形式で与えられることが多く，簡単な Lisp 処理系を組み込むことによって，開発期間やコストの低減が期待できる．しかしながら，アプリケーション開発者が Lisp 処理系実装のノウハウを備えていることは期待できず，限られた開発期間内に Lisp 処理系を実装するのは困難である．また，アプリケーションの要求を満たすように既存の Lisp 処理系を改造するのも，一般的には容易でない．

本稿では，Java 言語によって開発するアプリケーションに組み込んで使用することを主要目的として設計・開発した Lisp ドライバを紹介する．設計にあたって特に重視したのは次の項目である．

(1) Lisp 処理系の実装ノウハウを持たない Java プログラマにも機能の追加・削除・変更が容易に行えること．
(2) Java で開発したソフトウェア部品を扱うための機能を容易に組み込めること．
(3) コンパクトな実装であること．
(4) 高度な Lisp プログラム開発支援ツールを備える必要はないが，デバッグのために最低限必要な機能は備えること．
(5) 高性能である必要はないが，性能が極端に悪くないこと．

項目 (1) から，処理系記述言語は必然的に Java になる．Lisp の組込み関数を Java で容易に定義できれば，

† 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University

Java の部品を利用するためのインターフェイスは容易に構築できるので，項目 (2) も満たすことができる．また，Java の豊富なクラスライブラリを利用すれば，コンパクトかつ許容範囲の性能を有する処理系の実現が期待でき，残りの３つの項目も満たせる可能性がある．

しかし，Java で記述すれば項目 (1) を必ず満たせるとは限らない．かつて筆者らは，Java で記述した「ぶぶ」[19),20)] という Scheme 処理系を開発したことがある．「ぶぶ」は実行性能を最優先して設計したために，これを改造することは，一般の Java プログラマには困難である．例えば，car という単純な組込み関数でさえ，これを定義するメソッドは，次のように複雑である．

```
public static void Lcar(BCI bci) {
  Object x = bci.vs[bci.vsbase + 1];
  if (!(x instanceof List))
    throw SE.notList(x);
  bci.acc = ((List) x).car;
}
```

コードの詳細な説明は省略するが，このコードを理解するためには，「ぶぶ」のマルチスレッド機能，スタック構造，引数・返り値の受渡し方法など，実装に関するさまざまな知識が要求される．今回開発した処理系では，はるかに理解しやすい次の形で定義している．

```
public static Object car(List x) {
  return x.car;
}
```

実装の詳細とは無関係に，car という関数の機能（リストを受け取り，その car 部の値を返し，その値は任意の Lisp データである）を，忠実に Java コードに置き換えたものである．メソッドの修飾子は public static でなければならないが，実装の詳細を知らなくても，その理由は容易に想像できる．このような関数定義の枠組のみならず，項目 (1) を満たすためには，処理系実装上のさまざまな局面で工夫が必要であるが，それらについては，後の節で適宜述べることにする．

項目 (2) については，上記の関数定義の枠組があれば，Java で記述された部品を呼び出すための組込み関数が容易に定義できることは明らかであろう．一般的には，Lisp から Java への呼出しのみならず，逆に Java から Lisp への呼出しも望ましいことがある．Java のクラスのサブクラスを Lisp で定義したいような場合である．実際，上述の「ぶぶ」はそのような機能を備えている[11)]が，これを実現するためには，処理系の巨大化が避けられず，Java アプリケーション組み込み用の処理系には必ずしも必要ないと判断した．Lisp コードから使用したいサブクラスがあれば，Java で定義すればよい．

処理系をコンパクトにするために，Java ランタイムの持つ諸機能と豊富なクラスライブラリを有効に利用した．例えば次の機能を，処理系実装に直接利用した．

- メモリ管理とごみ集め
- 標準的なクラスで，Lisp のデータ型としてそのまま利用できるもの（2節参照）
- 入出力関係の諸機能（7節参照）

これらを有効に利用することによって，処理系のソースコードはわずか約 3500 行，100K バイト程度に収まっている．ソースコードのみならず，Java コンパイラが生成するクラスファイルのサイズも，コンパクトな処理系を達成するための重要な要因である．クラスファイルの構造上，小さなクラスほど，クラスファイルのサイズは相対的に大きくなる．そこで，処理系の可読性を低下しない範囲で，処理系を実装するクラスを極力少なく抑えるように努力した．現時点では，実装用のクラスは 13 個であり，そのサイズは合計 74K バイト程度である．

処理系の実行性能は，項目 (1)〜(3) よりも優先度が低い．したがって，他の項目を満たす範囲で，できるだけ性能向上を図るにとどまっている．しかしながら，10節で示すように，Java アプリケーション組み込みの処理系としては許容できる範囲に収まっている．

Lisp プログラムのデバッグ機能としては，次のものを備えている．

- 適切なエラーメッセージ表示機能
- エラー検出時に至る関数呼出しの履歴を表示するバックトレース機能
- 関数ごとに，呼び出し時の引数と結果の値を表示する関数トレース機能

関数トレース機能については，従来の Lisp 処理系の実装技法が利用できるが，メッセージ表示とバックトレースの実装は必ずしも自明でない．Java の機能を多用しているので，エラーの多くは Java のランタイムあるいはクラスライブラリが検出する．これらが生成するエラーメッセージは，Java でプログラム開発する際には有用であっても，Lisp プログラムのデバッグのためには情報不足であったり，メッセージの意味が理解できなかったりする．例えば，型の不整合が検出されたとき，Java が生成するエラーメッセージには Java のクラス名が記述されているが，処理系が表示するメッセージには Lisp のデータ型名を記述する必要がある．バックトレースについても同様である．エ

ラー検出時に Java が生成する例外オブジェクトには Java メソッドのバックトレース情報が含まれているが，これをそのまま表示しても，Lisp プログラムのデバッグには役に立たない.「ぶぶ」における上述の car の定義のように明示的に型検査を行ったり，関数呼出しのたびにバックトレース生成のための情報を保存するなどの方法で実装することは不可能ではないが，処理系コードの可読性，コンパクトさ，実行性能の点で望ましい方法ではない．可読性，サイズ，性能を悪化させない方式を考案し，実装している．

このように設計・実装した処理系はすでに完成している．以下では，本処理系の実装方法のうち，上記の目標を実現するために特徴的なものを 3節から 7節で報告する．これらを理解するための予備知識として，処理系の言語仕様とデータ表現を次節で述べる．8節では本処理系の利用方法を述べ，9節で他の処理系との比較を行う．最後に 10節で，処理系の性能について触れる．

2. 言語仕様とデータ表現

Lisp 処理系がサポートする言語の仕様は，本研究にとって本質的ではないが，今回開発した処理系は，IEEE Scheme[5),8)] のほぼフルセットを採用している．IEEE Scheme に準拠していないのは，次の 3点である．

- 継続（continuation）は，それを生成した call/cc 関数がリターンした後は呼び出せない．つまり，escape procedure[9)] として機能し，Common Lisp[16)] における catch&throw のような非局所的脱出には利用できるが，コルーチンを実現することはできない．これは，Java のランタイムスタックをヒープに退避するための処理系非依存の方法がないためである．
- 末尾再帰（tail-recursive）呼出しの最適化は行わない．これも，Java のランタイムスタックを直接操作する方法がないためである．
- 文字列は immutable であり，既存の文字列中のある文字を別の文字で置き換えることはできない．これは，文字列を Java の String オブジェクトで表現しているためである．String オブジェクトをラッピングするクラスを定義すれば，mutable な文字列は容易に実現できる．しかしそのための処理系の肥大や実行時オーバーヘッドに見合うだけの価値があるとは思えない．また，Java プログラマにとっては文字列が immutable であることは既成の事実である点も考慮した．

以下本稿で Lisp の言語仕様に言及するときは，特に断らない限り，IEEE Scheme のものを指すことにする．

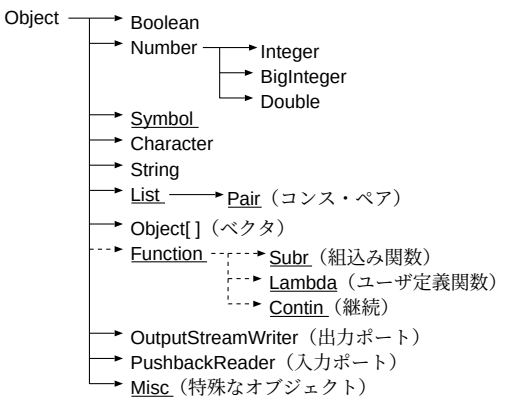

図 1 Lisp データを表現するクラス
Fig. 1 Classes representing Lisp objects

Lisp データを表現するために使用した Java のクラスを図 1に示す．矢印はクラス階層を表し，始点に位置するクラスが，終点に位置するクラスのスーパークラスであることを意味する．下線を引いたクラスは，処理系実装のために定義したクラスであり，その他は Java の標準的なクラスである．各クラスの表現する Lisp データを括弧内に記すが，自明なものは省略している．

BigInteger は任意精度整数，いわゆる bignum を表現する．bignum をサポートするのはオーバースペックのように見えるかもしれないが，アプリケーションによっては，bignum をビットベクタとして利用することがあり，サポートすることにした．Integer は 32 ビットの fixnum を表現する．bignum の演算結果が 32 ビット整数として表現できる場合は，結果は fixnum に正規化される．逆に fixnum 演算の結果が 32 ビットに収まらないときは，結果は自動的に bignum となり，整数演算がオーバーフローを起こすことはない．

List クラスは，空リスト '()' をコンス・ペアと同様に扱うために導入した．このクラスのインスタンスは，空リストだけである．Lisp の伝統に従って，空リストの car と cdr の値は，空リスト自身とした．空リストは，List クラスの final static 変数である nil に格納されており，List 以外のクラスからは，List.nil として参照される．

Function は実際はクラスではなく，インターフェイスであり，Subr，Lambda，Contin は，Function を実装（implements）する．その理由については，後述する．Misc クラスは，入力関数が入力ポートの終わりに達したときに返す eof-object など，特殊なオブジェクトを表現するものである．このクラスのインスタン

スで Lisp データとして使われるのは eof-object だけであり，他のインスタンスは，処理系が内部的に使用する．

3. インタープリタ

Lisp のコンパイル技術は成熟しており，変数や関数が型情報を持たない言語にもかからわず，効率の良いコード生成が可能である．しかし，処理系作成者以外が，Lisp コンパイラを改造することはきわめて難しい．特に問題となるのが，if や do といった特殊フォーム（Scheme の用語では syntax）である．コンパイラは特殊フォームを解析し，目的コードに変換する．したがって，特殊フォームの仕様変更や追加を行うためには，コンパイラの改造が必要となり，本研究が目指す目標が達成できない．このために，コンパイラを実装することによる効率向上は断念し，S 式（評価の対象となる Lisp データ）を解釈しながら実行するインタープリタによる実行方式を採用した．

本処理系のインタープリタは，eval というメソッドで実装されている．

```
static Object eval(Object expr, Env env)
```

S 式と環境（environment，局所変数の束縛情報）とを受け取り，与えられた環境の下で S 式を評価し，その結果を返す．eval への第 1 引数と評価結果は，任意の Lisp データ（Object）である．

S 式が特殊フォーム（特殊フォームの名前で始まるリスト）であれば，その cdr（先頭要素を除いた残りのリスト）と環境とを引数として，特殊フォームを実行するための Java メソッドを呼び出す．これらのメソッドは，前述の組込み関数 car の定義と同様の，直感的に理解しやすい形式で定義されている．例えば，条件分岐を行う if 式は，次のように実装されている．

```
public static Object Lif(Object c,
Object e1, Object e2, Env env) {
  if (eval(c, env) != Boolean.FALSE)
    return eval(e1, env);
  else if (e2 != null)
    return eval(e2, env);
  else
    return List.nil;
}
```

このメソッド定義は，if 式

(if c e_1 e_2)

の実行を，その仕様に従って忠実に記述したものである．まず eval を再帰的に呼び出して条件 c を評価し，結果が真（Boolean.FALSE 以外）であれば，再度 eval を呼び出して e1 を評価する．条件 c の評価結果が偽であれば，e2 を評価する．e2 は省略可能であり，省略された場合は空リストを返す．Java 言語の null は，Lisp データにはなり得ず，この定義のように，引数が与えられなかったことを示すような場合に用いられる．なお，メソッド名の “Lif” は，特殊フォーム名と同じ “if” としたいところだが，“if” は Java の予約語であり識別子としては使えないので，“Lif” としている．

特殊フォームと，それを実装するメソッドをリンクするには，defSpecial というメソッドを使う．if 式の場合であれば次の式を実行する．

```
defSpecial("Eval", "Lif", "if", 2, 1,
false);
```

Eval クラス（10節参照）で定義されている Lif というメソッドが，特殊フォームの if 式を実装し，if 式は少なくとも 2 引数を受け取り，省略可能な引数をもう 1 つ受け取ることができることを表している．defSpecial への最後のパラメータは，特殊フォームが任意個の引数を受け取れるかどうかを表す．if 式の場合は高々 3 引数しか受け取れないので，このパラメータに対する実引数は，false である．任意個の引数を受け取れる begin 式

(begin e_1 $\cdots$ e_n)

の場合は，次のように指定する．

```
defSpecial("Eval", "begin", "begin", 0,
0, true);
```

不定個の引数は，1 本のリストとしてメソッドに受け渡される．したがって begin は次のように実装できる．

```
public static Object begin(List es, Env
env) {
  Object val = List.nil;
  while (es != List.nil) {
    val = eval(es.car, env);
    es = (List) es.cdr;
  }
  return val;
}
```

関数呼出しの実行は，環境を受け渡す必要がないことを除けば，特殊フォームの実行とほとんど同じである．唯一の相異点は，S 式中の引数をそのまま受け渡すのではなく，それらを評価した結果を受け渡す点である．本処理系では，引数の評価と受け渡しのために，古典的な evlis 方式を採用している．すなわち，引数を評価した結果を 1 本のリストとして関数に受け渡す．

evlis 方式は，n 個の引数を受け渡すために n 個のコンス・ペアを消費するので，実行効率の点からは好

ましくない．引数の評価結果をスタックに積んでゆき，すべての引数の評価が終わったら，引数が積まれているスタック位置を指定して関数を呼び出すほうが効率面では優れている．実際，近年のLisp処理系の大多数は，スタックを使って引数を受け渡している．しかしこの方法は，処理系の可読性や拡張性を低下させる要因となる．Javaで処理系を実装する場合，引数の評価結果を１つずつランタイムスタックに積んでいくことはできない．そこで，配列などを使って，引数受け渡しのためのスタックを用意することになる．このスタックはJavaの実行機構とは無関係なので，スタックポインタを常に正しい値に維持するのは，処理系の責任である．引数の評価中にエラーが発生した場合に，リカバリ後のポインタ値を設定しなおすコードを，処理系の随所に埋め込む必要がある．また新しい制御機能を追加する場合には，ポインタ値の維持を常に念頭に置いておく必要があり，必要なら再設定のためのコードを埋め込まねばならない．スタックやスタックポインタが大域的な制御情報であるのに対して，evlis方式が使用するリストは局所的なデータであり，引数評価の途中でエラーが発生すれば単に廃棄されるだけである．したがって性能面では劣るもののの，我々がより重視する可読性・拡張性の面では，はるかに優れている．

スタック関係に限らず，本処理系は，大域的な制御情報を排除するように，設計・実装されている．その一例が，特殊フォームへの環境の明示的受け渡しである．大域変数に現時点の環境を格納しておき，適宜参照・更新を行えば，実行効率は向上しそうである．しかし，この大域変数の値を常に正しく維持するのは簡単ではない．処理系の可読性・拡張性を高めるためには，実引数として明示的に受け渡すほうが望ましい．

evlis方式を採用したもう１つの理由は，特殊フォームの実行と組込み関数の呼出しが，同じ機構で実現できることである．特殊フォームの場合は，S式のcdrを受け渡してメソッドを起動する．組込み関数の場合は，引数の評価結果をリストにして受け渡す．どちらの場合もリストが受け渡され，あらかじめ指定されたパラメータ情報（特殊フォームの場合はdefSpecialで指定された情報）に基づいて，実装用メソッドへの実引数として受け渡される．この機構は，特殊フォームの場合に環境も受け渡す点を除けば，特殊フォームと組込み関数の呼出し手順はまったく同じである．実際，本処理系では同一のルーチン（次節で述べるSubr.invoke）を使用している．

組込み関数と，それを実装するメソッドをリンクするには，defというメソッドを使う．関数carの場合は，次のように指定する．

```
def("List", "car", "car", 1, 0, false)
```

defへの引数の意味は，defSpecialの引数とまったく同じである．defとdefSpecialは，実装する対象がそれぞれ組込み関数と特殊フォームである点だけが異なっている．

以上述べたように，特殊フォームと組込み関数が酷似しており，実行機構が共有できることは，本研究の過程で見出された大きな発見であった．この特性によって，処理系コードの理解が容易になる（片方が理解できれば，もう片方も簡単に理解できる）とともに，コンパクトな処理系の実現にも貢献している．

本処理系が採用した上記の実装方式は，さらに別の意味でコンパクトな処理系実現に貢献している．同一の実装用メソッドが，複数の組込み関数を実装したり，処理系の内部ルーチンとしても利用できる場合があるからである．極端な例が，リストを受け取って，それをベクタに変換するメソッド

```
public static Object[] list2vec(List x)
```

である．このメソッドは，

```
def("List", "list2vec", "list->vector",
1, 0, false)
```

と指定することによって，リストをベクタに変換する組込み関数list->vectorを実装するとともに，

```
def("List", "list2vec", "vector", 0, 0,
true)
```

と指定することによって，任意個の引数を受け取って，それらを要素とするベクタを生成する組込み関数vectorも実装する．この２つの組込み関数は，Lispレベルでは異なった動作をする．

```
>(list->vector '(1 2 3))
#(1 2 3)
>(vector 1 2 3)
#(1 2 3)
```

しかし引数の受け取り方法が異なるだけで，内部的には同一のメソッド list2vecで実装されている．

さらにlist2vecは，通常のサブルーチンとして，バッククオートマクロを実装するために内部的にも使われている．ベクタをいったんリストに変換してからマクロ展開した後，展開結果をベクタに戻すために使われている．また，read関数がベクタリテラルを読み込むときにも，いったんリストとして読み込んでからベクタに変換するために使われている．実装用メソッドが，処理系の実行機構に依存しないように定義できるために，通常のサブルーチンとしても利用できるので

ある．

4. 関数呼出し

2節で述べたように，本処理系は，関数として組込み関数，ユーザ定義関数，継続の３種類をサポートしている．これらを実装する Java クラスの Subr，Lambda，Contin は，それぞれのインスタンスを関数として呼び出すための instance メソッド

```
public Object invoke(List args)
```

をクラスごとに定義している．ここで args は，呼び出し時にインタープリタが生成した引数リストである．6節で述べるように，Contin は例外クラス（RuntimeException のサブクラス）として定義している．Java は多重継承を許さないので，３種類の関数を統一的に扱うための Function は，３つのクラスが実装（implements）するインターフェイスとして定義した．クラスではなく，インターフェイスとしたことによって，コーディングに際して若干の制約が生じるが，ほとんどの場合，Function はスーパークラスのように使われている．例えば，あるデータが関数かどうかを判定するための組込み述語 procedure?は，次のように実装されている．

```
public static Boolean procedurep(Object
x) {
  return x instanceof Function ?
         Boolean.TRUE : Boolean.FALSE;
}
```

本節では，Subr クラスの invoke（Subr.invoke）について詳細に議論することにし，最後に Lambda.invoke について簡単に触れる．Contin.invoke については，節を改めることにする．

Java のメソッドとして実装された組込み関数を，Lisp 処理系から呼び出すために，Java の提供する reflection 機能を利用している．前節で述べたように，組込み関数の初期化には，def を使う．

`def(`*cs*，*mt*，*fn*，*nreq*，*nopt*，*auxp*`)`

は，まず *cs* という名のクラスで定義されている *mt* という名の public メソッドを検索する．次に，Subr クラスのインスタンスを生成し，その中に，見つかったメソッド（Method クラスのインスタンス）と，def への残りの引数を格納する．この Subr オブジェクトが，*fn* という名の関数データを表現する．最後に，*fn* という名の記号を（もし存在していなければ）生成し，その値スロットに，生成した Subr オブジェクトを格納する．

Java の reflection 機能では，Method オブジェクトとして取り出したメソッド M を呼び出すためには，M が受け取る引数の個数と同じ長さの配列（以下,「引数配列」とよぶ）を用意し，実引数を順に格納して受け渡さなければならない．このように受け渡された実引数が，M の定義中の引数の型と整合するかどうかは，reflection 機能が自動的に検査する．この検査に合格してはじめて，メソッド M が呼び出される．引数配列の長さは，def による指定によって，次の式で初期化時にあらかじめ計算しておくことができる．

nreq `+` *nopt* `+ (`*auxp* `? 1 : 0)`

引数配列は，関数呼出しのための invoke が呼び出されてから，実際にメソッド M が呼び出されるまでの一時的な格納場所なので，関数呼出しごとに生成する必要はない．そこで本処理系では，初期化時に長さの異なる引数配列をいくつか用意しておき，適切な長さの引数配列を使って実引数を受け渡すことにしている．Subr.invoke は，与えられた引数リストの各要素を，*nreq*，*nopt*，*auxp* の値に従って引数配列に格納していく．この過程で，正しい個数の引数が関数に渡されたかどうかが検査できる．もし引数の個数が正しくなければ例外を発生する．つまり，適切なエラーメッセージを格納した Java の例外オブジェクトを生成して，throw する．

reflection 機能が実引数の型検査によってエラーを検出した場合は，IllegalArgumentException クラスの例外を発生する．しかしこの例外オブジェクトは Java が生成したものであり，格納されているエラーメッセージは，

```
argument type mismatch
```

と，いささか不親切である．Lisp プログラムのデバッグには，最低限でも「何番目の引数が，これこれの型でなかった」といった情報がエラーメッセージに含まれてほしい．Subr.invoke が，reflection 機能を起動する前に自分で型検査を行ってエラーメッセージを生成することは不可能ではない．しかしこの方法では，同じ型検査が，Subr.invoke と reflection とで重複して行われることになる．このジレンマを解決するために，Subr.invoke は次の方法を採用している．まず，自分では型検査を行わないで，とにかく reflection 機能を起動する．もし IllegalArgumentException クラスの例外が発生したらそれを捕捉（catch）し，自分で型検査を行って適切なエラーメッセージを含んだ例外を発生する．この方法によって，関数の呼出しにオーバーヘッドをかけることなく，適切なエラーメッセージが生成できる．

このような，Java のエラーメッセージから Lisp の

エラーメッセージへの変換は，実装用メソッド中のキャストがエラーを検出した場合にも必要となる．例えば，与えられたリストxのn番目の要素を取り出す組込み関数list-refは，次のメソッドで実装されている．

```
public static Object nth(List x, int n) {
  while (--n >= 0)
    x = (List) x.cdr;
  return x.car;
}
```

もし引数xが真のリスト（cdrをたどっていくと，空リストで終わるデータ構造）でなければ，3行目のListクラスへのキャストでエラーが検出される．このとき，Javaのランタイムは，ClassCastExceptionクラスの例外を発生するが，そのオブジェクトには，キャストエラーを起こしたオブジェクトのクラス名が格納されているだけである．Javaプログラムをデバッグしているときなら，十分とはいえないまでも，必要最低限の情報は与えられている．Javaメソッドのバックトレースを見てエラーが起きたメソッドを特定し，そのメソッドの本体におけるキャストの場所を探せば，おおよその問題箇所は特定できる．しかしLispプログラムをデバッグする場合には，次の3つの問題がある．まず，キャストは実装上の概念なので，「ClassCastExceptionが発生した」と表示されても，Lispプログラムをデバッグしているプログラマは戸惑うであろう．また，エラーメッセージのクラス名は，実装上のクラス名であり，Lispデータの型名ではない．さらに，そのクラス名が，誤って与えられたオブジェクト（上の例では，x.cdrの値）のクラス名なのか，期待されていたクラス（上の例では，List）の名前なのかが分からない．このために，Subr.invokeは，実装用メソッドの実行中にClassCastExceptionが発生したらそれを捕捉し，クラス名を実装用のクラスの名前から，Lispレベルの型名に変更したメッセージを生成し，それを格納した例外を発生する．もし可能であれば，「Listが期待されているのにSymbolが与えられた」といったメッセージが望ましいが，期待されている型名は，捕捉した例外オブジェクトから特定することはできない．そこで現時点では，単に「unexpected Symbol object」といったメッセージを生成するにとどまっている．しかしこの変換だけでも，もとのエラーメッセージに較べれば，Lispプログラムのデバッグには役に立つ．

この他にもJavaのランタイムはさまざまな例外を発生するが，上にあげた2種類以外に対しては，Subr.invokeは特別な処理を行っていない．それらのエラーメッセージをすべて調べたわけではないが，これまでのところ，特に不都合はないようである．例えば，

```
(/ 1 0)
```

を評価するとArithmeticExceptionが発生するが，そのメッセージは

```
/ by zero
```

であり，Lispのエラーメッセージとしても通用する．

次に，ユーザ定義関数を呼び出すためのLambda.invokeについて簡単に触れておく．言語仕様としてSchemeを採用しているので，ユーザ定義関数は基本的にラムダ式で定義される．ラムダ式が評価されると，Lambdaクラスのインスタンスが1つ生成され，ラムダ式中のパラメータ情報（記号またはリスト）と関数本体（S式のリスト）が，生成時点の環境とともに格納される．Lambdaオブジェクトに対してinvokeメソッドが適用されると，与えられた引数リストをパラメータ情報と照合することによって，各引数の値を求め，Lambdaオブジェクト生成時点の環境に追加してゆく．この新しい環境の下で，本体を実行する．この一連の処理中にClassCastExceptionが発生した場合は，Subr.invokeと同様のメッセージ変換を行う．ただし，reflectionを起動するわけではないので，IllegalArgumentExceptionに対する特別な処理は行わない．

5. バックトレース

Lispプログラムをデバッグするときに，エラーが発生した関数の呼出しに至る関数呼出しの履歴を表示してくれるバックトレース機能は，きわめて有効である．本格的なLisp処理系では，ランタイムスタック中の関数フレームのリンクをたどることによって，バックトレースを表示することが多い．しかし，本処理系のようにJavaで記述した場合は，Javaのランタイムスタックを直接参照することができない．したがって，いつエラーが発生してもバックトレースを表示できるような機構が，Javaの実行機構とは別に必要となる．

まず考えられる方法は，関数呼出しごと（本処理系の場合，invokeメソッドの呼出しごと）に，呼び出される関数の名前を保存していく方法である．このためには，関数呼出しの履歴を保存するためのスタックを使用するのが一般的な実装法であろう．しかし，3節で述べたように，スタックのような大域的な制御情報は，本処理系の目的にそぐわない．たとえスタックの使用が容認されるとしても，バックトレースはエラーが発生しないと表示されないので，スタックに積まれた情報のほとんどは，利用されないで破棄される．そのような情報を，関数呼出しごとにスタックに積むのは無

駄である．以上の理由から，本処理系では，Java の例外処理機能を有効に利用した，実行時オーバーヘッドをともなわない実装方法を考案し，実装した．

処理系起動時に，backtraceToken という特殊な例外オブジェクトを１つ用意しておく．また，Subr.invoke と Lambda.invoke は，それらの実行中に発生した例外を，すべて捕捉するようにする．例外を捕捉したときに，エラーメッセージを表示するのか，あるいは呼び出した関数の名前をバックトレースの一部として表示するのかを判別するために，backtraceToken を使用する．もし捕捉した例外オブジェクトが backtraceToken 以外であれば，invoke は例外オブジェクトの情報に基づいてエラーメッセージを表示し，自分が呼び出した関数の名前を，バックトレースに現れる最初の関数名として表示する．その後，backtraceToken を throw する．この結果，invoke が捕捉した例外オブジェクトが backtraceToken であれば，他の invoke がすでにエラーメッセージを表示し，現在はバックトレースの表示中ということになる．そこで，backtraceToken を捕捉した invoke は，自分が呼び出した関数の名前だけをバックトレースの一部として表示し，backtraceToken を throw しなおす．このようにして，一連の invoke 呼出しが，関数名を表示しながら次々と巻き戻される．最後には，処理系のトップレベルに到達し，そこでバックトレースの表示は終わる．例として，次の関数を呼び出すことを考える．

```
(define (fact x)
  (if (zero? x)
      (/ 1 0)
      (* x (fact (- x 1)))))
```

Lisp の関数例としてよく使われる，階乗を計算する関数である．ただし，引数がゼロのときにゼロによる割算の例外が発生するように，意図的に間違えている．この関数を

```
(fact 3)
```

と呼び出すと，次のようにエラーメッセージとバックトレースが表示される．

```
ArithmeticException: / by zero
Backtrace: / < if < fact < if < fact <
if < fact < if < fact < top-level
```

トップレベルから fact が再帰的に４回呼び出され，４回目の呼出しの if 式の実行中に，割算の関数 ‘/’ が呼び出されて，ゼロによる割算が検出されたことが分かる．１行目のエラーメッセージと，２行目先頭の “`Backtrace: /`” までが，‘/’ を呼び出した invoke による表示である．この invoke は backtraceToken を throw し，直前の，if 式を実行中の（if 式を実装する Java メソッドを呼び出した）invoke がこれを捕捉し，続く “`< if`” を表示して，backtraceToken を throw しなおす．それをさらに，直前の，最後に fact を呼び出した invoke が捕捉し，続く “`< fact`” を表示し，backtraceToken を throw しなおす．このようにして最後は処理系のトップレベルに到達し，バックトレースの最後の “`< top-level`” を表示する．

invoke における例外の捕捉は，次の try-catch 文によって実装できる（実際の実装では，次節で述べる継続の実行機能が追加される）．

```
try {
  関数の呼出し処理
} catch (Throwable e) {
  if (e != backtraceToken) {
    エラーメッセージを表示し，
    バックトレース表示を開始する．
  } else {
    呼び出した関数の名前を，
    バックトレースの一部として表示する．
  }
  throw backtraceToken;
}
```

よく知られているように，Java のランタイムシステムは，まったくオーバーヘッドなしに，try-catch 文の本体（上の例では，関数の呼出し処理）を実行することができる．したがって本処理系は，関数呼出しにオーバーヘッドをかけることなしに，例外が発生したときには適切なエラーメッセージとバックトレースを表示することができる．

6. 継　　続

継続は，組込み関数の call/cc が生成する．call/cc は１引数の関数を受け取り，生成した継続を引数として呼び出す．2節で述べたように，本処理系では，IEEE Scheme の継続を完全に実現することは断念し，escape procedure としての機能を実現する．つまり，

```
(call/cc f)
```

によって生成された継続 c は，関数 f の実行中に限り，呼び出すことができる．継続 c が呼び出されると，f の実行は直ちに終了し，c への引数が，call/cc の値として返される．f の実行中に c が呼び出されないで，f が正常にリターンしたときは，f の返す値が，call/cc の値となる．いずれの場合も，call/cc がリターンした後では，継続 c を呼び出すことはできない．

ある継続 c が呼び出されたときに，それを生成した

call/ccがまだリターンしていないかどうかを検査する必要がある．もしリターンした後であれば，前節で述べた方法でエラーメッセージを表示し，バックトレースを表示しながら，トップレベルに戻る．また，cを生成したcall/ccへ一挙にリターンするためには，必然的にJavaの例外処理機能を使うことになる．以上の考察から，本処理系では，継続を次のように実装した．

まず，継続を例外オブジェクトとして表現し，継続の呼出しは，その継続をthrowすることによって実現する．throwされた継続は，それを生成したcall/ccだけが捕捉することにする．つまり，継続のクラスContinは一種の例外クラス（具体的には，RuntimeExceptionのサブクラス）とし，call/ccを，基本的に次のように実装する．ここでinvoke1は，関数を1引数で呼び出すためのメソッドであり，継続contを引数として関数fを呼び出している．

```
public static Object callcc(Function f) {
  Contin cont = new Contin();
  try {
    return f.invoke1(cont);
  } catch (Contin c) {
    if (c == cont)
      return c.value;
    else
      throw c;
  } finally {
    cont.canCall = false;
  }
}
```

個々の継続オブジェクトには，それを生成したcall/ccがまだリターンしていないかどうかを記憶するスロットcanCallを設ける．その初期値はtrueであり，call/ccがリターンする前に，それが生成した継続オブジェクトのcanCallを必ずfalseにする．継続を呼び出すためのContin.invokeは，このスロットを検査してからthrowすればよい．

前節では，エラーメッセージまたはバックトレースを表示するために，Subr.invokeとLambda.invokeがすべての例外を捕捉すると述べた．しかし継続オブジェクトだけは，エラーが発生したわけではないので，捕捉してはならない．このために，前節にあげた疑似コードは，実際には次のようになる（“...”の部分は，前節の疑似コードと同じなので省略する）．

```
try {
  関数の呼出し処理
} catch (Contin c) {
  throw c;
} catch (Throwable e) {
  ...
}
```

2番目のcatch節は，継続も含めすべての例外を捕捉して，エラーメッセージやバックトレースを表示してしまうので，継続をいったん捕捉してthrowするだけのcatch節をその前に設け，2番目のcatch節に到達できないようにしているのである．

この節で紹介した継続の実装方法は，Common Lispなどにおけるcatch&throw機能を実装する場合にも適用可能である．catch&throwの場合は，任意のLispデータをthrowできる．そのデータと同一のデータを指定してcatcherを設定したcatch式から脱出する．継続の場合は，継続どうしを比較して脱出先を決定したが，catch式に指定されたデータをcatcher設定の際に継続オブジェクトの中に格納しておき，データどうしを比較することによって脱出先を決定するようにすれば，catch&throw機能が実現できる．

7. 入 出 力

図1に示したとおり，本処理系では，出力ポートをOutputStreamWriterクラスで，入力ポートをPushbackReaderクラスでそれぞれ表現している．いずれも，Javaの標準のクラスである．前者は，Javaの出力ストリームをラッピングし，指定された文字コードで日本語文字を出力ストリームに出力する能力がある．入力ポートについては，入力ストリームをInputStreamReaderでラッピングして日本語文字の入力能力を与え，それをさらにPushbackReaderでラッピングすることによって，直前に読み込んだ1文字をunreadする能力を持たせている．ラッピングのためにポートへの入出力処理の効率は低下するが，日本語文字の入出力能力は，それに見合うだけのメリットがあると判断した．ラッピングするだけで，日本語文字を入出力できるLisp処理系が実現できるのは，Javaで実装することの大きな利点である．

入力ポートからLispデータを読み込むためのread関数の実装にあたっては，改造ができるだけ容易に行えるように配慮した．Lispの S 式はLispのデータなので，read関数が容易に改造できるということは，Lispの構文を容易に変更できることを意味する．改造を容易にするための基本的なアイデアは，read関数の振る舞いをLispレベルで変更できるCommon Lispの機能を，実装レベルで採用することである．

Common Lispのread関数は，文字ごとに構文属

性を与え，これを使って入力文字列からトークンを切り出し，一定の規則に従ってそれを Lisp データに変換する．ただし「マクロ文字」の属性が与えられた文字は特別扱いされる．これらの文字にはリーダマクロとよばれる関数が与えられ，その文字で始まる入力文字列の処理は，リーダマクロが行う．文字の構文属性を変更したり，Lisp 関数をリーダマクロとして登録する機能を Lisp レベルで提供しており，Lisp プログラムが read 関数の振る舞いをカスタマイズすることが可能になっている．

本処理系が前提とする用途を考えた場合，Lisp レベルで read 関数の振る舞いを変更する必要はなさそうである．read 関数を改造するのは Java プログラマなので，Java レベル，すなわち処理系の記述言語のレベルで容易に改造できればよい．そこで，Common Lisp の read 関数の機能を，実装レベルで採用した．例えば，開き括弧 '(' には「マクロ文字」の属性を与え，入力文字列の先頭が開き括弧であれば，リストを読み込むためのコードを実行する，といった具合に read 関数が実装されている．

read 関数の実装にあたっても，Java の機能を極力利用することによって，処理系をコンパクトにする努力を行っている．read 関数は，入力文字の構文属性に基づいてトークンを切り出した後，それが数値を表現するものであれば数値データに変換し，そうでなければ記号データに変換する．本処理系では，数値データを表現するために，Integer，BigInteger，Double の３つの Java クラスを使用している．そのそれぞれに，パージングのための機能が用意されているので，それを利用することにした．これによって，トークンから Lisp データへの変換は，基本的に次のような簡単なコードで実現できている．

```
try {
  return makeInt(Integer.parseInt(s));
} catch (NumberFormatException e) {}
try {
  return new BigInteger(s);
} catch (NumberFormatException e) {}
try {
  return new Double(s);
} catch (NumberFormatException e) {
  return Symbol.intern(s);
}
```

s が，パージングの対象となっているトークン文字列である．まず Integer としてパースを試みる．成功すれば int 型の整数が返ってくるので，それを Integer オブジェクトに変換したものを返す．失敗すれば NumberFormatException が発生するが，throw された例外オブジェクトは無視して，次に BigInteger としてのパースを試みる．それにも失敗したら次は Double としてパースを試みる．すべてのパースに失敗したら，s を名前とする記号を（まだ存在していなければ）生成し，それをパージングの結果として返す．ここで，Integer，BigInteger，Double の順序は重要である．Java の仕様では，Integer としてパースできる文字列は，BigInteger，Double としてもパースでき，BigInteger としてパースできる文字列は，Double としてもパースできるからである．

数値データのパージングを Java の機能に依存するということは，数値データのテキスト表現として Java の構文を採用したことになる．幸いなことに，Java の数値表現は，Lisp の数値表現として使っても違和感がない．むしろ，本処理系を利用する Java プログラマにとっては，Java とまったく同じ構文を使えるほうが便利であろう．

Lisp データを出力ポートに書き出すための write 関数の実装にあたっては，read 関数と整合性がとれるように配慮した．write 関数の出力結果を read 関数で読み込んだ結果は，もとと「同じ」Lisp データになってほしい．記号データはもとと同一の記号になってほしいし，数値データはもとと同じクラスで，もとの数値と等しい数値になってほしい．本処理系は Java の数値表現を採用したので，出力も Java の機能を使うことによって，入出力表現の整合性をとっている．すなわち，write 関数が数値データを出力するときは，Java の toString メソッドを使って文字列に置き換え，その内容を出力している．

記号データを出力するときは，read 関数が記号として読み込めるように，必要ならエスケープ文字を付加する（具体的には，記号名を縦棒 '|' で囲む）必要がある．エスケープが要らないのは，

- 記号名が１個のトークンとして読み込まれ，
- それが数値を表現するものではない

場合である．前者は，記号名を走査して文字の構文属性を調べればよい．後者は Java の数値表現に依存するので，read 関数と同様の方法で判断している．つまり，Integer，BigInteger，Double としてパースを試み，そのすべてが失敗すれば「数値を表現しない」と判断する．このために，エスケープの要・不要の判断は時間がかかる上に，パースが成功すればその数値クラスのインスタンスが生成されてしまうので，メモリ効率も悪い．そこで本処理系では，記号データを出力

したら，その出力結果を文字列として記号データ内に保存しておき，同じデータが再度出力されるときは，その文字列の内容を出力することにしている．これによって，エスケープ要・不要の判断は，記号データごとに高々１回しか行われないことになる．

8. 利 用 法

本処理系は，Java アプリケーションに組み込まれて利用されることを目的にしているが，通常の Lisp 処理系として単独でも利用可能である．処理系の main メソッドは Eval クラスで定義されており，このクラスを JVM に指定することによって，処理系が起動する．

```
% java Eval
```

main メソッドは，実装用クラスをロードすることによって処理系を初期化した後，read-eval-print ループ（以下では，REPL と略す）を呼び出す．REPL から抜けると，処理系の実行は終了する．

```
public static void main(String argv[]) {
  initializeSystem();
  readEvalPrintLoop();
}
```

REPL は，標準入力から S 式を１つ読み込み，それを表示して結果を標準出力に表示する．これを，入力の終わりに達するまで繰り返す．この過程で例外が発生した場合は，エラーメッセージを表示する（トップレベルで発生した場合）か，あるいは “< top-level” と表示してバックトレース表示を完了する．

```
public static void readEvalPrintLoop() {
  for(;;)
    try {
      IO.print("\n>");  // プロンプト表示
      Object expr = IO.read(null);
      if (expr == IO.eofObject) break;
      IO.println(topLevelEval(expr));
    } catch (Throwable e) {
      if (e != backtraceToken)
        エラーメッセージを表示する.
      else
        バックトレース表示を完了する.
    }
}
```

本処理系を Java アプリケーションに組み込む場合は，さまざまな利用形態が考えられる．もっとも典型的なのは，Lisp で記述したプログラムをロードし，S 式を与えて実行する形態であろう．これは，次のコード列によって実現できる．

```
(1) Eval.initializeSystem();
(2) Eval.loadProgram(ソースファイル名);
(3) Object value = Eval.runProgram(S 式);
```

ここで，ソースファイル名と S 式は文字列である．loadProgram は，組込み関数の load を実装するメソッドを呼び出すための API である．また runProgram は，与えられた文字列のための入力ポートを生成し，そのポートから S 式を読み込んで評価し，結果を返す．これらの API は，次のように定義されている．

```
public static void loadProgram(String s) {
  try {
    IO.load(s, null, null);
  } catch (Throwable e) {...}
}
public static Object runProgram(String s) {
  try {
    return topLevelEval(IO.read(IO.openIn
putString(s)));
  } catch (Throwable e) {...}
  return null;
}
```

これらの定義における catch 節の本体は，REPL のものと同じである．これによって，REPL と同様に，実行中に例外が発生した場合に正しく処理できる．

3節で述べたように，本処理系では，組込み関数は実装用メソッドと def（特殊フォームの場合は defSpecial）式のペアによって定義されている．それぞれの def 式は，実装用クラスのロード時に実行されるように，static 文の中に与えられている．また，実装用メソッドと，その def 式の対応がとりやすいように，処理系のソースでは，両者が連続して記述されている．例えば，関数の car は，実際のソースでは次のように定義されている（def 式は簡略形が使われている）．

```
static { Subr.def("List", "car", 1); }
public static Object car(List x) {
  return x.car;
}
```

組込み関数を削除したり変更する際には，組込み関数の名前でソースを検索すれば，def 式と実装用メソッドが容易に見つかる．これらを削除すれば，その組込み関数は処理系から抹消されるし，これらを変更すれば，組込み関数の動作が変わる．例えば，関数の car が空リストを受け付けないようにしたければ，上の定義の２行目を次のように変更すればよい．

```
public static Object car(Pair x) {
```

前述のように，本処理系はスタックなどの大域的な制

御情報をいっさい使っていないので，削除や変更を安心して行える．

組込み関数や特殊フォームを追加したければ，既存の定義にならって def 式や実装用メソッドを与えればよい．例えば，Common Lisp の when 式に相当する特殊フォームを追加したければ，次のコードを Eval クラスに記述すればよい．

```
static { Subr.defSpecial("Eval", "when",
"when", 1, 0, true); }
public static Object when(Object c, List
es, Env env) {
  if (eval(c, env) != Boolean.FALSE)
    return begin(es, env);
  else
    return List.nil;
}
```

既存の Java 部品を Lisp から利用したいときにも，組込み関数を追加する必要がある．例えば，Java アプリケーションでビットベクタのクラス BV を定義しており，Lisp プログラムからもビットベクタを利用したい場合を考える．ビットベクタは，本処理系が本来対象とする Lisp オブジェクトではないが，BV が Object のサブクラスである限り，ビットベクタを Lisp 変数に代入したり，Lisp 関数に引数として受け渡すことができる．したがって，ビットベクタを処理する組込み関数を追加すれば，他の Lisp オブジェクトと同様に Lisp プログラムからビットベクタを利用できるようになる．ビットベクタ用の組込み関数の定義例をあげる．VB クラスの instance メソッドとして，ビットごとの and を求める操作が定義されているときに，それに対応する Lisp 関数を定義する例である．

```
static { Subr.def("BV", "BVand", 2); }
public static BV BVand(BV x, BV y) {
  return x.and(y);
}
```

さらに，ビットベクタを Lisp のオブジェクトとして入出力したければ，まず read 関数を改造する．Lisp では，ビットベクタのような付加的なデータの場合は，入力処理をリーダマクロとして定義できるように構文を定めるのが慣例である．例えば，‘#*’ の後にビット列（0 と 1 の列）を続ける，という構文を採用したと仮定しよう．7節で述べたように，本処理系の read 関数は，内部的に Common Lisp の実装方式を採用している．入力文字列が ‘#’ で始まる場合は，sharpSignMacroReader というメソッドが残りの読み込みを担当する．このメソッドの本体では，switch 文を使って，‘#’ に続く文字によって入力処理を振り分けている．この switch 文に，次が ‘*’ の場合の処理を追加すればよい．一方，Lisp オブジェクトを出力するための write 関数は，特に指定がない場合は，オブジェクトに対して toString メソッドを適用し，得られた文字列を表示する．したがって，VB クラスの toString メソッドを，入力と整合するように定義すると良い．もし VB クラスの変更が許されないなら，write 関数の定義を書き換えることによって対処できる．

9. 他の処理系との比較

Java で記述された Lisp 処理系で，ソースコードが公開されているものに，HotScheme[4]，Skij[17]，Kawa[3]，Jscheme[2]，SISC[13] がある．本節では，これら５つの処理系の実装方法を，本処理系と比較・検討する．いずれも言語仕様は Scheme がベースであり，HotScheme は Scheme 風の独自の言語，Skij，Kawa，Jscheme は IEEE 仕様[5),8)] にほぼ準拠，SISC は R^{5}RS 仕様[10)] に完全準拠，となっている．Lisp プログラムの実行方式は，HotScheme と Skij が S 式を直接実行するインタープリタ方式であり，他の３つはコンパイラ方式である．Kawa は JVM に変換するが，Jscheme と SISC は独自の内部コードに変換した後，内部コードのインタープリタによって実行する．

本処理系の実装上の最大の特徴は，処理系の実行方式に依存せずに，特殊フォームと組込み関数を定義している点にある．これによって，Lisp 処理系実装のノウハウを持たない Java プログラマにも，処理系の改造を容易に行えることになる．コンパイラ方式を採用している３つの処理系では，特殊フォームをコンパイラが変換するので，特殊フォームを改造するためには，コンパイラの構造を理解する必要がある．このうち Jscheme では，特殊フォームの大多数を Scheme のマクロとして定義しているので，マクロ定義を変更すれば，それらの特殊フォームの改造は比較的用意である．しかし，マクロとして定義できない基本的な特殊フォームはコンパイラが直接に変換する必要があり，それらの改造は容易でない．インタープリタ方式を採用している HotScheme と Skij では，特殊フォームは S 式を受け取り，S 式の構文チェックは個々の特殊フォームの定義にまかされている．このために特殊フォームの定義は複雑になりがちであり，構文チェックの洩れが生じやすい．実際，これらの処理系では構文チェックを怠っている箇所が随所にあり，次のように不適切なエラーメッセージを表示することがある．

```
HotScheme:
```

```
  HotScheme >> (if)
  error: Arg not a list:  in car of: #f
Skij:
  skij> (if)
  SchemeException: Can't eval null
```

本処理系では，構文チェックのほとんどはインタープリタが行うので，特殊フォームの定義に構文チェックのコードを入れる必要がなく，エラーメッセージもインタープリタが適切に生成する．

```
>(if)
RuntimeException: too few arguments to if
```

組込み関数の実装にあたっては，いずれの処理系も，基本的に引数の評価結果をリスト（あるいはベクタ）として受け渡している（ただしKawaでは，いくつかの典型的な呼び出しパタンの場合にJavaスタックを介して受け渡すように工夫されている）．このため，組込み関数は任意のLispオブジェクトを引数として受け取り，引数の型チェックは，個々の組込み関数の定義にまかされている．このために，明示的な型チェックのコードを挿入すると定義が複雑になり，逆に挿入を怠ると，不適切なエラーメッセージを生成することになる．

本処理系では，staticメソッドとして定義した組込み関数を，Javaのreflection機能を使って呼び出しているが，５つの処理系のうちreflectionを使っているものはない．reflectionの実行性能が最大の理由であろう．HotScheme，Skij，Kawaは，組込み関数ごとにクラスを用意し，関数呼出しのためのメソッドによって組込み関数を実装している．このためにクラスファイルの個数と合計サイズが大きくなっている（表2参照）．一方，JschemeとSISCは，組込み関数ごとに固有の番号を割り当て，実行時にswitch文で分岐させている．この場合は処理系は比較的コンパクトになるが，組込み関数の追加・削除のためには番号の割り当てを考慮する必要があり，慎重な作業が要求される．

HotSchemeとSISCは，バックトレース機能を持たない．Kawaのバックトレース機能はJavaレベルのバックトレースなので，Lispプログラムのデバッグには使いにくい．SkijとJschemeはLispレベルのバックトレース機能を持ち，本処理系と同様に，Javaの例外処理機能を使って実装している．本処理系は関数呼出しを巻き戻しながら関数名を表示していくのに対して，これらの処理系では，１つの関数呼出しを巻き戻すたびに，バックトレース表示用のJavaオブジェクトを１つ生成する．それらをリンクして保持することによって，呼び出された順に関数をバックトレース表示することが可能である．また，バックトレース用オブジェクトに呼出し時の引数情報などを格納でき，単に関数名だけでなく，詳しい呼出し履歴を表示することができる．本処理系よりも高度な機能を提供しているが，筆者の経験では，関数名だけの単純なバックトレースのほうが使いやすい．またそのほうが，コンパクトな処理系を目指す本処理系の場合には適している．

継続を，本処理系と同じくescape procedureとして実装しているのは，Skij，Kawa，Jschemeである．いずれもJavaの例外処理機能を使って実装している．Javaにはescape procedureを（許容できる性能内で）実装するための機能が他にはないので，これは当然である．SkijとJschemeは，継続を呼び出す際に，その継続を捕捉できるかどうかをチェックしていない．このため，捕捉できない継続を呼び出したときは，処理系のトップレベルに制御が移動し，バックトレースを表示することができない．Kawaは本処理系と同様の技法で，捕捉できるかどうかのチェックを行っている．本処理系との本質的な相異点は，関数の一種である継続オブジェクトと，それが呼び出されたときにthrowされる例外オブジェクトを分けている点である．call/ccによって継続が生成されるときには，それぞれのオブジェクトを１つずつ生成し，例外オブジェクトを継続オブジェクトに格納する．継続オブジェクトが呼び出されたときは，格納されている例外オブジェクトを取り出して，それをthrowする．しかし，２つのオブジェクトの生存期間はまったく一致するので，本処理系のように，１つのオブジェクトで実現することが可能である．Kawaでは，継続オブジェクトのクラスと，継続用の例外オブジェクトのクラスを別々に用意する必要があるが，本処理系では，１つのクラス（Contin）で実現できている．

Kawaのread関数は，本処理系と同様に，入力文字の構文属性に基づいてパースを行っているので，５つの処理系のなかでは，もっとも解読しやすい．しかし，数値データのパースは独自のコードで実装しているので，巨大で難解である．５つの処理系のうち，Javaの数値パース機能を利用してread関数を実装しているのは，Jschemeだけである．しかしJschemeでは，write関数による出力形式が，read関数と整合していない．つまり，記号を出力するときに，read関数が記号として読み込めるためのエスケープ文字をまったく付加しない．本処理系では，出力に際しても，Javaの数値パース機能を利用することによって，write関数とread関数との整合性を保っている．

表 1 実装用クラスとそれらのサイズ
Table 1 Implementation classes and their sizes

クラス名	ソースファイル 行数	ソースファイル バイト数	クラスファイル バイト数
Char	256	7,786	6,618
Contin	61	1,307	1,516
Env	66	1,655	1,888
Eval	576	18,377	12,006
Function	10	172	257
IO	767	23,128	15,446
Lambda	61	1,479	2,259
List	449	11,666	8,353
Misc	14	192	342
Num	728	22,462	12,267
Pair	10	168	287
Subr	219	6,714	6,692
Symbol	235	6,541	6,009
合計	3,452	101,647	73,940

10. 評　　価

処理系を実装するために定義したクラスと，それらのサイズを，表1に示す．Eval クラスには，インタープリタと特殊フォームの多くが定義されている．Char は文字と文字列に関する操作を，IO は入出力操作を，Num は数値計算のための組込み関数を，それぞれ定義している．Env は環境を表現するためのクラスである．その他のクラスについては，図1を参照されたい．

処理系全体でソースコードにして約3,500行，100K バイト程度であり，クラスファイルのサイズは合計74K バイトである．コンパクトな処理系であることは間違いない．前節にあげた５つの処理系および「ぶぶ」とのサイズ比較を，表2に示す．組込み関数などを Scheme で定義している処理系があるので，ソースファイルについては，Java と Scheme の両方のデータを掲載する．処理系によって言語仕様が異なるので，単純に比較できないが，本処理系がきわめてコンパクトであることが分かる．

実行性能を評価するために，Gabriel のベンチマークテスト[6] を実行した．その結果を，表 3に示す．テストは，Solaris 7 を OS とする Sun Ultra 60（UltraSPARC-II 360MHz）で行った．使用した Java ランタイムは JDK 1.2 であり，Java 上の時間計測には，java.lang.System.currentTimeMillis() を使用した．比較のために，C 言語で記述された Kyoto Common Lisp[18] (KCL) のインタープリタを使って実行した結果，Scheme プログラムを JVM にコンパイルして実行する Kawa の実行結果，S 式をインタープリタが直接実行する Skij の実行結果をあげる．

KCL インタープリタと比較すると，本処理系は平均して５倍程度の時間を要している．JVM インタープリタによって実行されているのが最大の要因だと考えられるが，ctak，fprint，fread の３つのプログラムで特に差が大きいのは，Java の例外処理機能（ctak の場合）と入出力機能（fprint と fread の場合）の性能が影響しているようである．Kawa と比較すると，本処理系は３～５倍程度の時間を要している．KCL とは異なり，Kawa と本処理系は同じ JVM インタープリタで動作するので，テストプログラムによる実行時間比のばらつきが小さい．実行時間の差は，インタープリタ方式とコンパイラ方式の相違と考えてよいであろう．インタープリタ方式の Skij と比較すると，本処理系はほぼ同等の性能である．本処理系はその開発目的のために，Java の reflection 機能を使って，特殊フォームや組込み関数を実行している．reflection 機能は実行時に複雑な型チェックを行うために，当初から実行性能の面では期待が持てなかった．しかし，実行時のチェック機構の弱い Skij とほぼ同等の性能を有することは，高く評価してよいであろう．「高性能である必要はないが，極端に性能が悪くないこと」という当初の目標は達成できたと考えている．

11. おわりに

Java アプリケーションに組み込んで利用するための，Lisp 処理系の設計と実装について報告した．Lisp 処理系の実装に精通していない Java プログラマにも，機能の変更・削除・追加が容易に行えるように，実装上のさまざまな工夫を行った．しかし Java アプリケーション開発の現場でどの程度有効であるかは，今後の使用経験を積まなければ，正確には判断できない．

ここで報告した処理系はすでに完成しており，筆者の Web ページでフリーソフトウェアとして公開している[21]．

本研究を始めたのは，COINS プロジェクト[14] から Lisp 処理系についての相談を受けたことがきっかけである．COINS プロジェクトは，コンパイラの共通インフラストラクチャの構築を目指している．インフラストラクチャの一部となる実験的なコンパイラを Java で開発しており，その中間コードやターゲットマシン記述[1] は，Lisp の S 式と同様の形式で表現されている．このために，処理プログラムを Lisp で記述できれば，開発期間を大幅に短縮できる可能性がある．実験的なコンパイラに組み込んで使え，簡単に仕様変更ができるような Lisp 処理系がないものか，またそのような処理系を開発するとしたらどの程度の期間と労力が必要か，といった相談を受けた．Lisp 処理系の

表 2 処理系のサイズ比較
Table 2 Comparison of Implementation Sizes

処理系	Java ソースファイル			クラスファイル		Scheme ソースファイル		
	ファイル数	行数	バイト数	ファイル数	バイト数	ファイル数	行数	バイト数
本処理系	13	3,452	101,647	13	73,940	0	0	0
HotScheme	114	5,674	139,363	121	134,811	0	0	0
Skij	41	5,136	151,841	173	280,427	53	3,818	122,439
Kawa	135	12,306	338,927	160	429,886	18	1,541	48,582
Jscheme	51	7,609	266,338	55	228,150	42	7,794	262,642
SISC	90	12,448	408,201	96	223,001	38	10,796	381,559
ぶぶ	55	16,018	470,255	69	309,675	8	4,509	170.232

表 3 ベンチマーク実行結果（単位は秒）
Table 3 Benchmark results (in seconds)

テスト	本処理系	KCL	Kawa	Skij
ctak	15.754	1.700	8.329	16.826
deriv	8.569	1.660	2.550	6.425
div				
(ite)	6.505	0.890	0.829	10.587
(rec)	8.445	1.720	0.909	8.535
fprint	0.106	0.010	0.118	0.981
fft	27.310	7.600	11.192	エラー
fread	0.851	0.030	0.506	0.059
puzzle	76.258	13.410	20.547	96.751
tak	4.399	1.486	1.021	4.172
takr	4.663	1.790	1.561	5.017
tprint	0.050	0.010	0.096	1.138
traverse				
(init)	69.476	15.870	20.598	132.938
(run)	363.169	101.520	56.675	373.436
triangle	1067.892	230.320	219.743	1000.534

利用を検討している阿部正佳氏（東京大学），阿部氏とともに議論に加わっていただいた鈴木貢氏（電気通信大学）をはじめ，COINS プロジェクトのメンバの方々に感謝したい．小宮常康氏（京都大学）には，本処理系と比較するための処理系のインストールに協力いただいた．この場をかりて礼を述べたい．

参 考 文 献

1) 阿部正佳, 藤波順久, 中田育男, 萩谷 昌己: LIR: COINS プロジェクトの低水準中間言語, 第 39 回情報処理学会プログラミング研究会 (2002).
2) Ken Anderson, Tim Hickey, and Peter Norvig: The Jscheme Web Programming Project, jscheme.sourceforge.net/jscheme/mainwebpage.html (2002).
3) Per Bothner: Kawa, the Java-based Scheme System, www.gnu.org/software/kawa/ (2002).
4) Gene Callahan, Brian Clark, Rob Dodson, and Prasad Yalamanchi: HotScheme, www.stgtech.com/HotScheme/ (1997).
5) Clinger, W. C. and Rees, J.: Revised^4 Report on the Algorithmic Language Scheme, MIT AI Memo 848b, MIT (1991).
6) Richard P. Gabriel: Performance and Evaluation of Lisp System, MIT Press Series in Computer Schience, MIT Press, Cambridge, MA (1985).
7) Gosling, J., Joy, B., and Steele, G.: The Java Language Specification, Addison-Wesley (1996).
8) IEEE Standard for the Scheme Programming Language, IEEE (1991).
9) Takayasu Ito and M. Matsui: A Parallel Lisp Language PaiLisp and Its Kernel Specification, Lecture Notes in Computer Science 441, Springer-Verlag, pp. 22–52 (1995).
10) R. Kelsey, W. Clinger, and J. Rees: Revised^5 Report on the Algorithmic Language Scheme, Higher-Order and Symbolic Computation, Vol. 11, No. 1 (1998).
11) 窪田貴志, 湯淺太一, 倉林則之, 八杉昌宏, 小宮常康: Java 上の Scheme 処理系「ぶぶ」における単一のクラスローダを用いたオブジェクトシステムの実装, 情報処理学会論文誌, 42 巻 SIG 7(PRO 11) 号, pp.57-69 (2001).
12) Robert McFarlane, Camillus P. McElhinney, and Bob McFarlane: Using Autolisp With Autocad, John Wiley & Sons (1999).
13) Scott G. Miller and Matthias Radestock: SISC for Seasoned Schemers, sisc.sourceforge.net (2002).
14) 中谷俊晴, 加藤吉之介, 佐々政孝, 脇田建: コンパイラ・インフラストラクチャにおける SSA 形式最適化プロトタイプシステムの実装, 日本ソフトウェア科学会大会論文集, 第 18 回, 3D-2 (2001).
15) Richard M. Stallman, Dan Laliberte, and Bill Lewis: The Gnu Emacs LISP Reference Manual, Free Software Foundation (1991).
16) Guy L. Steele: Common Lisp the Language, Second Edition, Digital Press (1990).
17) Michael Travers: Skij: Scheme in Java for Interactive Debugging and Scripting, alphaworks.ibm.com/tech/Skij (1999).
18) Taiichi Yuasa and Masami Hagiya: Ky-

oto Common Lisp Report, Technical report, Teikoku Insatsu Publishing, Kyoto (1985).
19) Taiichi Yuasa: An object-oriented Scheme System Bubu and Its Facility for Parallel Computation, International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, pp. 101–121 (1999).
20) 湯淺太一: Java で Scheme を書いてみれば，第42回プログラミング・シンポジウム報告集, pp.27–36 (2001).
21) 湯淺太一: Java アプリケーション組み込み用の Lisp ドライバ，www.yuasa.kuis.kyoto-u.ac.jp/~yuasa/jakld (2002).