携帯電話で使えるiアプリすぷ

湯淺 太一　鵜川 始陽

携帯電話端末にダウンロードして利用できる Lisp 処理系を紹介する．本処理系は，実時間ごみ集めの研究実験の副産物として開発された．最近の携帯端末には，アプリケーションをダウンロードして利用できるように，KVM をベースにした極めて小型の Java VM を備えているものが少なくない．我々はこれらの携帯端末において高性能の実時間ごみ集めを実現するために，既開発および新規開発の実装技術を試験実装してきた．性能評価を行うために，メモリ管理能力のベンチマークプログラムが豊富にそろっている Lisp を採用することにした．具体的には，Java で記述した小型の Lisp 処理系である JAKLD を KVM 用に改造し，その上で Lisp ベンチマークプログラムを実行し性能評価を行った．JAKLD は，Java アプリケーションに組み込んで利用することを目的として開発した Lisp 処理系である．それ自体，小さく設計された処理系ではあるが，KVM の機能制約は厳しく，さらに小型化する必要があった．このように改造した超小型の処理系に，携帯端末固有の API を駆使することによって，実際に携帯端末で動作させることが可能となった．

We present a Lisp system that can be downloaded to and used on mobile phones. This system was developed as a side-product of research experiment on real-time garbage collection. Many mobile terminals are now equipped with a tiny Java VM based on KVM so that the users can download application programs for their terminals. In order to realize high performance real-time garbage collection on such mobile terminals, we have been implementing various implementation techniques experimentally. We chose Lisp for evaluation, since there are many benchmark programs available in Lisp for memory management. We built a Lisp interpreter on KVM for mobile terminals by modifying a compact Lisp system JAKLD written in J2SE, and ran Lisp benchmarks for evaluation. JAKLD is designed to be used primarily as an embedded system in Java applications. JAKLD itself is a compact system, but we had to further reduce the size of the system in order to meet the limitation of functionality in KVM. The reduced system became available on mobile terminals by making use of APIs specially designed for mobile terminals.

1　はじめに

最近の携帯電話端末 (以下，携帯端末と略すことがある) には，一般ユーザが自分の端末にアプリケーションをダウンロードして利用できるように，Java [5] の実行系を備えているものが多い．PC には及ばないものの，端末本体の性能は向上を続けており，かなり高度なアプリケーションでも利用できる環境が整いつつある．常時持ち歩ける身近な計算資源であり，これを有効に利用してユーティリティの自作や端末のカスタマイズ，さらにはプログラミング教育などに利用できないか，という発想が自然に出てくる．そのような一歩進んだ利用形態の検討材料とするために開発し，一般に配信している i アプリすぷ (iαpplisp) を，本稿で紹介する．

本処理系は，一口で言えば，携帯端末で利用できる小型の Lisp 処理系であるが，最初から携帯端末での利用を目指して開発したわけではない．他のプロジェクト遂行のためのツールとして開発し，上記のような目的にも利用できそうだと分かったために，NTT DoCoMo の最新のほとんどの端末で利用できる i アプリ [9] として動作するように改造をほどこしたもので

A Lisp Interpreter iαpplisp That Runs on Mobile Phones.

Taiichi Yuasa, Tomoharu Ugawa, 京都大学情報学研究科, Graduate School of Informatics, Kyoto University.

コンピュータソフトウェア, Vol.24, No.4 (2007), pp.109–122.
[ソフトウェア論文] 2007 年 1 月 21 日受付.

ある．現時点では，NTT DoCoMo の i アプリ対応端末でしか動作を確認できていないので，i アプリすぷと呼んでいる．Java で記述された JAKLD [23] [24] という Lisp 処理系をベースにしており，Mobile JAKLD と呼ぶこともある．

開発の動機となったのは，我々が進めている実時間ごみ集め (GC) のプロジェクトである．これは，独自に開発した実時間 GC の諸技術を実際のプログラム実行系に適用し，実用性の検証を行うものである．その一環として，携帯電話端末などのモバイル端末における Java 実行系への応用を検討しており，そのような実行系のベースとして使われることの多い KVM/CLDC [15] に実時間 GC 技術の試験実装を進めていた．実装結果の性能評価を行うために，Java ではなく Lisp のベンチマークプログラムを利用することにした．Lisp には，GC をはじめとするメモリ管理性能評価のためのベンチマークプログラムが豊富にそろっていたためである．改造した Java 実行系の上で Lisp アプリケーションを実行するために，Java で記述した Lisp 処理系を必要とした．そこで，J2SE で動作していたコンパクトな Lisp インタープリタである JAKLD をベースにして，モバイル端末用の KVM/CLDC で動作するように修正を施したものが Mobile JAKLD である．

以下本稿では，まず次節で本処理系開発の経緯をもう少し詳しく述べる．続く 3 節で本処理系のベースとなった JAKLD の概要を紹介する．本処理系を理解する上で重要な特徴を述べ，その知識を前提として 4 節と 5 節で本処理系の詳細を述べる．6 節では本処理系のサイズや実行性能について触れ，最後に 7 節で今後の課題を与える．

2 開発の経緯

ごみ集め (garbage collection，GC) とは，アプリケーションプログラムが使わなくなったオブジェクトを，自動的に回収してメモリ領域を再利用する機構である．従来，GC 処理は新しいオブジェクトを生成する際に十分な空き領域がヒープに確保できなかった場合に起動され，GC の実行中はアプリケーションの実行が中断されていた．GC 処理のためのこのような停止は，実時間性や対話性を要求するアプリケーションにはきわめて都合が悪い．そこで考案されてきたのが，GC 全体の処理を小さな単位に分割し，アプリケーションの実行に伴って少しずつ実行するインクリメンタル GC(ソフト実時間 GC あるいは単に実時間 GC と呼ぶこともある) である．

実時間 GC は，アプリケーションの実行と交互に進行するために，単純に従来の GC 処理を分割しただけでは，まだ使用中のオブジェクトを誤ってごみとして回収する可能性がある．この問題を解決するために，我々はスナップショット GC [21] と，その改良方式であるリターンバリア機構 [22] などを提案し，さまざまなプログラミング言語実行系に試験実装し，その有効性の検証を行ってきた．そのうち特に目覚しい成果が，オムロン社が自社の組込み製品のために開発した JeRTy VM での実装 [12] である．JeRTy ではそれまで，別の実時間 GC 方式を採用していたが，スナップショット方式に移行することによって，実行時オーバーヘッドのある仮想マシン命令が，19 個から 3 個に減少した．このために，アプリケーションの実行時間が平均 30%短縮した．また実時間応答性能も飛躍的に向上し，GC 処理に伴う割込み禁止時間が，従来は 2.6〜20 ミリ秒 (アプリケーションごとの最大時間) だったのが，スナップショット方式にすることによって 0.2〜0.5 ミリ秒に低下した．スナップショット方式へ移行するためのコード変更量は少なく，GC 関係のコード全体が約 2400 行であるのに対して，追加が 110 行，削除 60 行，変更が 50 行だけであった．

これらの成功を受けて，現在，携帯端末や PDA に代表されるモバイル端末での Java 処理系への試験実装を進めている．現状では，実時間性を要求するアプリケーションの場合，GC による停止を避けたい部分の直前で，GC を強制的に実行する Java のメソッド `System.gc()` を呼び出すようにベンダが工夫しているようである．端末の機種が異なれば，`System.gc()` の呼び出しを挿入する場所も微妙に異なるため，挿入箇所の決定にはノウハウが必要といわれている．アプリケーションそのものは Java で記述されているので，端末が異なっても調整なしで動作するはずである．`System.gc()` の挿入が機種依存であるために，新規

機種で動作させるためには経費と時間を要している．

実時間 GC 技術の導入によってこの問題が解決することは明らかである．しかし一方で，メモリ量や CPU 性能が限られているモバイル端末において，実時間 GC 技術の導入がどの程度のオーバーヘッドを生じ，どの程度の実時間性を保証できるのかは自明でない．実際の携帯端末における Java VM は，ROM に収納されているために，その内容を変更することは一般の研究者には不可能である．そこで，これらの VM のベースとして使われることの多い KVM/CLDC [15] を改造して，実装方法の検討と性能評価とを行っている．KVM はモバイル端末用に開発された Java 実行系であり，数十 K バイト程度の小型の VM である．CLDC (Connected Limited Device Configuration) は，KVM とあわせて使用する基本的なクラスライブラリ群である．いずれも Sun Microsystems 社からソースプログラムが公開されている．

実時間化の改造を行った KVM/CLDC の性能評価を行うために，我々は Lisp のベンチマークプログラムを利用した．Lisp は GC をはじめとするメモリ管理技法の実験用プラットフォームとして使われることが多く，ベンチマークプログラムも豊富である．これらのプログラムを Java 実行系に対しても適用することによって，従来の Lisp 実行系との比較も可能になる．この方法を可能にするためには，Java で記述した Lisp 実行系があればよい．そこで，次節で概説する JAKLD [23] という処理系を利用することにした．

3 JAKLD

JAKLD は，Java アプリケーションに組み込んで使うことを目的として開発した Lisp ドライバである．その設計にあたっては，特に次の項目を重視した．

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．

```
public static Object car(List x) {
  return x.car;
}
public static Pair cons(Object x, Object y) {
  return new Pair(x, y);
}
public static Object setCar(Pair x, Object val)
{
  return x.car = val;
}
```

図 1 JAKLD における組込み関数の定義例

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，Java の部品を利用するためのインタフェー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，残りの三つの項目も満たすことができる．

JAKLD の組込み関数は，きわめて理解しやすいように記述されている．その一例として，組込みの Lisp 関数 `car`，`cons`，`set-car!` の定義を図 1 にあげる．Lisp の言語仕様を理解している Java プログラマにとっては，これらの定義に説明はまったく不要であろう．

処理系をコンパクトにするために，JAKLD では Java 実行系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ている．たとえば次の機能を，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
- 入出力関係の諸機能
- 例外処理機能
- reflection 機能

これらを有効に利用することによって，実装用のクラスはわずか 13 個にとどまり，ソースコードは約 3,500 行，100 K バイト程度に収まっている．また，Java

コンパイラが生成するクラスファイルのサイズも，合計 74K バイト程度と，きわめてコンパクトな処理系を実現している．JAKLD は現在，フリーソフトとして Web で公開されており [24]，いくつかの研究プロジェクト (例えば [17]) で言語機能の試験実装などのために利用されている．

3.1 言語仕様

JAKLD は，IEEE Scheme [3] [6] のほぼフルセットをサポートしている．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 [7] として機能し，Common Lisp [14] 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 の実行時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) 呼び出しの最適化は行わない．これも，Java の実行時スタックを直接操作する方法がないためである [19].
- 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が，そのための処理系の肥大化や実行時オーバヘッドに見合うだけの価値があるとは思えない．

Lisp データを表現するために使用した Java のクラスを図 2 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 J2SE 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スペックと思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．`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．

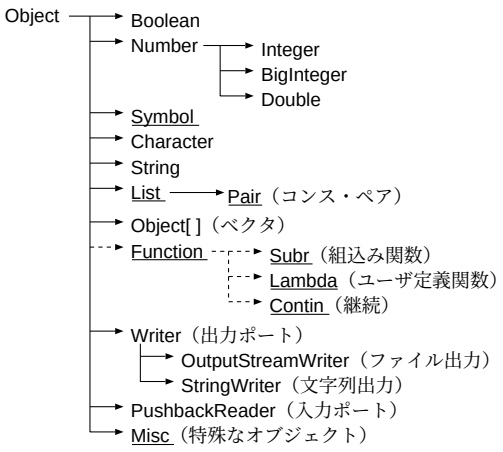

図 2 Lisp データを表現するクラス (JAKLD)

関数には，`Subr`(組込み関数)，`Lambda`(ユーザ定義関数)，`Contin`(継続) の 3 種類がある．前述のように JAKLD の継続は escape procedure として機能するので,「継続を呼び出す」ための処理は,「例外を投げる」ための処理と本質的に同じになる．そこで，`Contin` を例外クラス (`RuntimeException` のサブクラス) と定義することによって，Java の機能を効果的に利用して処理系をコンパクトに抑えている (3.3 節参照)．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，三つのクラスが実装 (implements) するインタフェースとして定義した．

JAKLD を Java アプリケーションに組み込んで使うときには，JAKLD 上の Lisp アプリケーションと Java アプリケーションが情報交換を行う必要が生じる．JAKLD のオブジェクトは Java のオブジェクトでもあるので，オブジェクトのレベルで情報交換を行

うことは可能である．加えて，文字列の形でメッセージとして情報交換したいという要請があった．Java アプリケーションに受け渡す文字列を Lisp アプリケーションが容易に生成できるように，出力ポートとしてファイル出力 (標準入出力を含む) に加えて文字列出力を利用できるようにした．

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．

3.2 インタープリタ

処理系の改造が容易に行えるように，JAKLD はコンパイラ方式を断念し，S 式 (評価の対象となる Lisp オブジェクト) を解釈しながら実行するインタープリタ方式を採用している．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 オブジェクト (Object) である．

S 式が特殊形式であれば，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，特殊形式を実行するための 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．たとえば，条件分岐を行う if 式は，図 3 のように実装されている．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 e_1 を評価する．条件 c の評価結果が偽であれば，e_2 を評価する．e_2 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 データにはなりえず，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

```
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;
}
```

図 3 JAKLD における特殊形式 if の定義

用いられる．なお，メソッド名の “Lif” は，特殊形式名と同じ “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．

特殊形式と，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 (6 節参照) で定義されている Lif というメソッドが，特殊形式の 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一つ受け取ることができることを表している．defSpecial の第 4 と第 5 のパラメータは非負整数であり，特殊形式が最低限必要とする引数の個数と省略可能な引数の個数をそれぞれ指定する．defSpecial への最後のパラメータは，特殊形式が任意個の引数を受け取れるかどうかを表す．if 式の場合はたかだか三つしか引数を受け取れないので，このパラメータに対する実引数は，false である．

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．例えば，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 への引数とまったく同じである．

関数呼び出しの実行は，環境を受け渡す必要がないことを除けば，特殊形式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．JAKLD 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．受け渡されたリストは，次節で述べる方法によって実装用メソッドに受け渡される．

3.3 関数呼び出し

3.1 節で述べたように，JAKLD 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．以下本節では，`Subr` クラスの `invoke`(`Subr.invoke`)，つまり組込み関数の呼び出し機構について解説する．

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．

4 J2SE から KVM/CLDC への移行

実時間 GC を実装した KVM/CLDC の性能評価を行うために，J2SE で動作していた JAKLD を KVM/CLDC で動作するよう改造を行った．KVM/CLDC は，基本部分の Java 言語仕様は J2SE と共通であるが，クラスライブラリに大きな制約がある．JAKLD の移行に特に影響したのは，次の相違である．

- reflection が使えない．
- `Number` と `BigInteger` クラスがない．
- 標準入力の `System.in` がない．
- 入出力関係の機能が大幅に削減されている．
- 文字関係の `Character` クラスと数値計算ライブラリを収めた `Math` クラスが縮小されている．

JAKLD の文字関係と数値計算関係の組込み関数の多くは，単純に `Character` あるいは `Math` クラスのメソッドを呼び出すようになっている．そのようなメソッドのいくつかが削減されたので，対応する組込み関数も削除した．`char-alphabetic?` や `atan` などがそうである．これらの組込み関数を自分で定義することは可能であるが，検討したどのベンチマークプログラムにも使われていないし，KVM/CLDC で未定義ということは実際の携帯端末でも利用されることは少ないと判断した．

`BigInteger` がないので，bignum のサポートは断念した．これまでの処理系開発の経験から，クラスライブラリを使わずにいちから bignum を実装すると，かなりの工数を要すると予想された．もともと JAKLD が bignum をサポートした理由は，ビットベクタとして利用できることであり，実時間 GC の性能評価を行うためのベンチマークプログラムには bignum は不要であった．また，処理系のサイズの面からも bignum のサポートは疑問であった．C 言語で記述した TUT Scheme [20] の場合，bignum 演算のためのコードは約 1800 行である．Java で記述すると，行数は若干減ると思われるが，演算コードだけで 1000 行以上にはなるだろう．後述のように，KVM/CLDC 上で動作する JAKLD(bignum なし) のソースは 4000 行弱である．ソースの行数が処理系

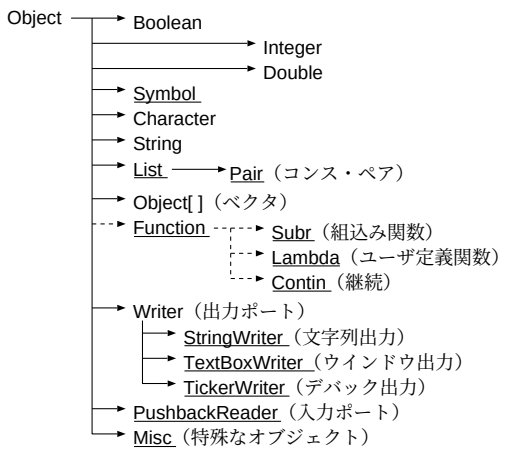

図 4 Lisp データを表現するクラス (本処理系)

のサイズに比例するとは限らないが，bignum をサポートすることによって行数が 25%増加すれば，処理系もかなり巨大化する．メモリが限られている携帯端末では大きな負担となる．

携帯端末でもビットベクタが必要かどうかは分からないが，もし必要となれば，bignum を実装して流用するよりも，新たにビットベクタのコードを追加するほうがはるかに工数が少なくて済む．TUT Scheme における上記 bignum 演算のうち，論理演算は約 800 行である．粗い見積もりであるが，論理演算が主となるビットベクタのコーディングは，算術演算も必要とする bignum と比べ，工数は半分以下と予測される．

bignum がなくなったので，数値データは，`Integer` による 32 ビット整数と `Double` による 64 ビット浮動小数点数だけとなった (図 4 参照)．数値データ全体をまとめた `Number` クラスが KVM/CLDC ではサポートされていないので，`Integer` と `Double` は `Object` クラスの直接のサブクラスである．このために，KVM/CLDC へ移行するにあたって，大きなコード変更を必要とした．例えば，オブジェクトが数値データであるかどうかを判定する組込みの Lisp 関数 `numberp` は，JAKLD では

```
public static
Boolean numberp(Object obj) {
  return obj instanceof Number ? T : F;
}
```

と定義されていたが，KVM/CLDC 用には

```
public static
Boolean numberp(Object obj) {
  return obj instanceof Integer ? T :
         obj instanceof Double ? T : F;
}
```

となる．また，組込み関数を定義するメソッドの引数や値の型に `Number` が使えなくなったので，やむを得ず `Object` に変更した．例えば，引数に 1 加える `1+` という関数は，JAKLD で

```
public static Number onePlus(Number num)
```

と定義されていたものが，

```
public static Object onePlus(Object num)
```

となった．プログラミング作法の観点からも望ましくない．

KVM/CLDC は，ファイル出力，文字列出力ポート，入力ポートを提供していないので，これらを自作した (図 2 と比較せよ)．ベンチマークテストの結果を表示するために標準出力 `System.out` へのファイル出力を用意したが，携帯端末では標準出力の概念が意味をなさないので，最終的に，次節で述べるウィンドウ出力用の `TextBoxWriter` と，デバッグ出力用の `TickerWriter` に置き換えた．標準入力がないので，実時間 GC のベンチマーク用にはファイルからの入力を，携帯端末用には，次節で述べるウィンドウ入力をサポートした．

J2SE の reflection が使えなくなったので，組込み関数の呼び出し (特殊形式の起動を含む) 機構をおおはばに変更した．しかし，JAKLD の処理系コードの可読性を損なわないように，組込み関数の定義は基本的にそのまま残すこととした．`car` の定義は図 1 のままであり，組込み関数の `car` をこの定義とリンクするための `def` メソッドの呼び出しも前節であげたものと同じである．

一般に Java では，switch-case 文の実行が高速である．専用の JVM 命令が用意されており，コンパイラはほぼ間違いなく，その命令を使ったコードを生成する．組込み関数を呼び出す機構は，この switch-case 文を使って書き換えた．まず，すべての組込み関数に通し番号 (関数番号) をふる．`def` は，引数として

```
Object doInvoke(int funnum, Object[] argV) {
  switch (funnum) {
  ...
  case 5: return List.car((List) argV[0]);
  ...
  case 71: return Eval.Lif(argV[0],
                           argV[1],
                           argV[2],
                           (Env) argV[3]);
  ...
  default:
    System.out.println("unknown function");
    return null;
  }
}
```

図 5 switch-case による関数ディスパッチ

与えられた関数名から，その関数の番号を探し，生成する Subr オブジェクトに番号を格納する．呼び出しの際は，switch-case 文でこの番号によってディスパッチし，関数を実装するメソッドを呼び出す．図 5 に，ディスパッチを行うメソッド doInvoke の概要を示す (紙面の都合上，クラスの参照やエラーメッセージは適宜簡略化している)．関数への引数受け渡しは JAKLD の evlis 方式をそのまま利用し，組込み関数の引数情報に基づいて，配列 (図 5 中の argV) に格納して doInvoke に受け渡す．引数配列の要素は Object クラスと宣言されているので，適宜キャストを付ける．例えば，関数番号が 5 の car の場合は，引数を List にキャストしてから List クラスの car メソッドを呼び出す．特殊形式の if(関数番号は 71) の場合は，最後の引数は if 式実行時の環境なので Env クラスにキャストしてから Eval クラスの Lif メソッドを呼び出す (if への最初の 3 引数は任意の Object なのでキャストは不要である)，組込み関数への引数の個数は引数リストから引数配列へ格納しなおす際に検査し，引数の型は，実装するメソッドを呼び出す際のキャストが検査する．これによって，関数呼び出し機構のコード量は増大したが，後述のように，実行性能はかなり向上した．

JAKLD には，200 以上の組込み関数 (特殊形式を含む) が定義されている．そのひとつひとつに関数番号をつけたり，doInvoke 内の実装用メソッド呼び出しを記述するのは，時間がかかる上に，手作業では間違う可能性も高い．そこで，J2SE の reflection 機能を利用してこの作業を自動化した．すでに述べたように，JAKLD は組込み関数を def メソッド (特殊形式の場合は defSpecial) を使って処理系起動時に登録する．JAKLD におけるこれらのメソッドを改造し，呼び出されたときに関数番号の割り当てと実装用メソッドを呼び出すコードをファイルに出力するようにした．def 式には実装用メソッドの引数の型情報が含まれていないが，これは reflection 機能を使って取り出すことができる．このように改造した JAKLD を J2SE 上で一度走らせ，KVM/CLDC 用への移行に必要なコードのかなりの部分を自動生成することに成功した．

以上のように KVM/CLDC 上で動作するように改造した JAKLD を使って，実時間 GC のプロトタイプを実装した KVM/CLDC で走らせ，Lisp のベンチマークテストを実行した．実時間 GC の性能評価のためには，実時間化したことによるオーバーヘッドの計測も必要であるが，アプリケーションの実時間応答性能がどの程度向上するかを測定する必要がある．これにはマイクロ秒単位の時間測定を必要とする．実際の携帯端末に近い PDA やスマートフォンでは難しいので，性能評価は PC で行った．その結果，比較的小さなオーバーヘッドで実時間性が大幅に向上することが確認されている．例えば Xeon プロセッサ (3.2GHz，512KB キャッシュ) で Boyer ベンチマークを実行した場合，GC によるアプリケーションの最大停止時間は，20.48 ミリ秒だったものが 11.3 μ秒と，約 1/2000 に抑えられている．現在は，プロダクトレベルの実時間 GC の実装を進めており，性能評価の詳細とあわせて，いずれ別の機会に報告したい．

5 携帯端末への移行

使ってみればすぐ分かることだが，携帯端末には，コンソールや Windows の DOS プロンプトに相当するものがない．テキスト入力は，入力専用のウィンドウに対して行う．それをソフトウェアが処理し，場合によっては結果を別のウィンドウに表示する．GC を実時間化した KVM/CLDC でベンチマーク用に改造した前述の JAKLD に，携帯端末のテキスト入出力

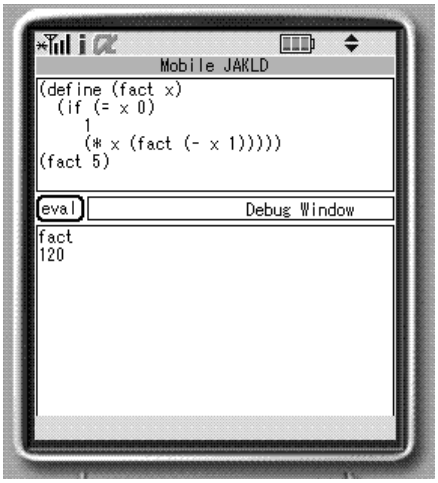

図 6 エミュレータ画面

機能を加えれば，携帯端末でも使えるはずである．実際の携帯端末の Java 実行系も，多くが KVM/CLDC をベースにしているからである．

ウィンドウ関連の機能は機種に依存するので，CLDC の範囲を超えて，携帯端末ベンダやキャリアごとに定めるクラスライブラリ群であるプロファイル (profile) によって定義する．Sun Microsystems 社からは MIDP (Mobile Information Device Profile) [16] というプロファイルが提供されており，多くの端末で標準的に使われている．しかし，NTT DoCoMo は独自のプロファイル仕様を使っており，それを DoJa [9] と呼んでいる．今回は，NTT DoCoMo の FOMA 端末をターゲットとしたので，DoJa プロファイルを使って，上記の入出力機構を実装した．DoJa 上で開発した KVM/CLDC のアプリケーションを NTT DoCoMo では i アプリと呼んでいるので，本処理系を i アプリすぷと呼ぶことにした．

開発には，DoCoMo から提供されている i アプリ開発用の SDK を利用した．この SDK は Windows PC 上で動作し，i アプリのビルド機能や，FOMA 端末のエミュレータを持っている．エミュレータには携帯端末の画面とボタンが備わっており，実際の携帯端末に近い動作をする．本処理系を実行しているときのエミュレータ画面を図 6 に示す．画面の上方に入力ウィンドウがある．階乗を計算する関数 `fact` の定義と，5 の階乗を計算する式 `(fact 5)` が入力されている．その下に eval と記したボタンがあり，これを押す (正確には，eval ボタンを選択し，「決定」ボタンを押す) と，入力ウィンドウの式を評価する．結果は，その下の出力ウィンドウに表示される．図では，定義した関数名 `fact` と，`(fact 5)` の計算結果である 120 が表示されている．

標準的なテキストウィンドウにはスクロール機能がなく，画面に収まらない部分は見えなくなる．eval ボタンを押すたびに結果が見えるようにするために，結果を表示する前に出力ウィンドウをクリアすることにした．通常はこの方法で不便がないが，デバッグ情報を表示する場合は不便である．ある式の評価中にデバッグ情報をプログラムが表示しても，式の評価が終わって結果を表示するときに，デバッグ情報がクリアされてしまうからである．そこで，出力ウィンドウとは別に，デバッグ出力用のウィンドウを提供することにした．図 6 で，eval ボタンの右にある横長のウィンドウがそうである．このウィンドウは ticker と呼ばれ，常に左方向にスクロールして，はみ出した部分は右端から現れる．やや奇妙な印象があるが，携帯端末画面の限られたスペースを有効に使うための選択である．

以上の機能を実現するためのトップレベルの概要を図 7 に示す．このコードは，処理の流れを理解しやすいように簡略化しており，実際のコードでは，ウィンドウの初期化や変数宣言などのコードが含まれている．`Entry` がトップレベルを定義するクラス (アプリケーション作成時にこのクラス名を登録する) であり，1 行目でこれが i アプリ (`IApplication`) を定義すること，2 行目でイベントを受け付けること (`ComponentListener`) が宣言されている．本処理系が起動されるとスレッドが一つ生成され，トップレベルである `Entry` クラスの `start` メソッドが呼び出される．この `start` メソッドでは，処理系画面 (`Panel`) を構成するウィンドウやボタンを設定してゆき，イベントを受け付けられる状態にしてから処理系画面を表示する．そして，標準的な入力・出力ポートとデバッグ出力ポートを準備しておいて，別スレッドで Lisp の評価器 (`Eval`) を起動する．評価器は，出力は出力

```
public class Entry extends IApplication
       implements ComponentListener {

  public void start() {

    Panel p = new Panel();

    p.add(new Label("Mobile JAKLD"));
    p.add(inbox = new TextBox(...));
    p.add(evalButton = new Button("eval");
    p.add(debugOut = new Ticker(...));
    p.add(outbox = new TextBox(...));

    p.setComponentListener(this);
    Display.setCurrent(p);

    consoleIn = new FifoReader();
    IO.currentInputPort
    = new PushbackReader(consoleIn);

    IO.currentOutputPort
    = new TextBoxWriter(outbox);

    IO.debugOutput
    = new TickerWriter(debugOut);

    (new Eval()).start();
  }

  public void componentAction
              (Component source,
                int type, int param) {
    if (source == evalButton && type
         == ComponentListener.BUTTON_PRESSED) {
      try {
         consoleIn.append(inbox.getText());
      } catch (IOException e) {}
    }
  }
}
```

図 7　携帯端末用トップレベルの概要

ウィンドウに直接表示するが，入力は consoleIn から受け取る．Entry クラスで定義されているもう一つのメソッドである componentAction が，入力ウィンドウの内容を consoleIn へ送る．このメソッドは，eval ボタンが押されると，その時点の入力ウィンドウの内容を取り出して consoleIn にアペンドする．評価器は，consoleIn からの入力を処理し終わると，次の入力があるまで待ち状態となる．このように入力を処理するスレッドと評価器のスレッドを分けることによって，評価器に変更を加えることなく，通常の Lisp 処理系に似た動作を行わせることが可能となる．

図 8　実機画面とダウンロードサイトの QR コード

i アプリの優れた点は，キャリアから認証を受けないでも，ユーザが自分のアプリケーションを自由にダウンロードして使えることであろう．エミュレータを使って開発したアプリケーションをダウンロードするには，携帯端末用のウェブページ (ユーザが自由に作ってよいので「勝手サイト」と呼ばれている) を用意し，クラスファイルをまとめた jar ファイルをアップロードしておけばよい．これを i モードで携帯端末にダウンロードすれば，直ちに実行できる．著者はこれら一連の処理を，本処理系を動かすために初めて利用したが，エミュレータでのテスト終了後，まったく問題なく実機でも動作した．図 8 に，本処理系が実機で動作している様子と，本処理系のダウンロードサイト [25] の QR コード [10] を示す．本処理系の jar ファイルのサイズは約 40K バイトであった．

6　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形式の多くが定義されている．Char

表 1 実装用クラスとそれらのサイズ

クラス名	.java 行数	.java バイト数	.class バイト数
Char*	247	7,404	6,543
Contin	60	1,247	1,458
Entry+	68	1,898	2,378
Env	65	1,595	1,886
Eval*	589	18,532	13,166
FifoReader+	93	1,592	1,461
Function	9	112	202
IO*	769	23,065	14,584
Lambda	60	1,419	2,204
List	447	11,557	7,727
Misc	13	132	259
Num*	507	15,003	9,202
Pair	9	108	212
PushbackReader+	59	1,076	1,000
StringWriter+	22	463	606
Subr*	583	22,741	22,620
Symbol*	232	6,453	6,179
TextBoxWriter+	27	522	808
TickerWriter+	27	518	804
合計	3,886	115,437	93,299

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．`Entry` は，本処理系のインタープリタと携帯端末ウィンドウを結合するために導入したクラスで，本処理系のメインクラスである．その他のクラスについては，図 4 を参照されたい．

表中，‘*’ は JAKLD から変更があったクラスを表し，‘+’ は新しく作成したクラスを表す．前者のうち，`Num.class` が JAKLD では 12,220 バイトだったものが，9,202 バイトに減少している．これは，サポートする組込み関数が減ったためである．また，`Subr.class` が JAKLD では 6,692 バイトだったものが，22,620 バイトと大幅に増加している．これは，J2SE の reflection 機能を利用できなかったためである．変更のあったその他のクラスについては，JAKLD とサイズはほぼ同じである．

処理系全体でソースコードにして約 3,900 行，115 K バイト程度であり，クラスファイルのサイズは合計 93K 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．JAKLD と，Java で記述した他の二つの Scheme 処理系とのサイズ比較を，表 2 に示す．Skij[18] も Kawa[2] も，IEEE 仕様[3][6] にほぼ準拠した処理系である．Skij が S 式を直接実行するインタープリタ方式であるのに対して，Kawa は JVM コードに変換して Java VM で直接実行する．いずれも，組込み関数などを Scheme で定義している部分があるので，ソースファイルについては，Scheme ソースのデータも掲載する．処理系によって機能が異なるので単純には比較できないが，本処理系がコンパクトであることが分かる．ただし，JAKLD と比較すると，既に述べた理由によってサイズが増加している．本処理系を実際に携帯端末にダウンロードする場合は jar ファイルにまとめたものを使うが，そのサイズは約 42 K バイトと小さい．

実行性能を評価するために，Gabriel のベンチマークテスト[4] のいくつかを，著者が普段使っている FOMA D901i[8] で実行した．CPU は SH-Mobile (200MHz 程度[†1]) であり，KVM/CLDC をベースにした JBlend[1] が利用できる．プロファイルは DoJa-4.0，ヒープサイズは 3M バイトである．実行結果を表 3 に示す．比較のために，本処理系を Windows PC(PentiumM 1.1GHz) 上の i アプリ用 SDK に付随するエミュレータ (KVM ベース) と JDK 1.5 で実行した結果と，同じ PC 上で同じ JDK 1.5 を使って JAKLD を実行した結果を掲載する．また，JAKLD，Skij，Kawa を Solaris 6 を OS とする Sun Ultra 60(UltraSPARC-II 340 MHz) 上の JDK 1.2.1 で実行した結果も掲載する．時間計測には，どの実行系でも `java.lang.System.currentTimeMillis()` を使用した．なお，どの Java 実行系でも，JIT を使わずに実行している．

実際の携帯端末における本処理系は，PC 上のものと較べると，10 数倍から 20 数倍程度の実行時間がかかる．CPU や周辺デバイス (メモリなど) の性能の差がそのまま実行時間に反映しているようである．boyer について特に差が大きいのは，このベンチマークのデータ消費量が多く，携帯端末の 3M バイトのヒープでは，頻繁にごみ集めが起動されているためであることが想像できる．しかし，我々の知る限り，ご

†1 正確なクロック数は公表されていない．メーカに問い合わせたところ，企業秘密とのことであった．

表 2　処理系のサイズ比較

	.java			.class		Scheme ソースファイル		
処理系	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	19	3,886	115,437	19	93,299	0	0	0
JAKLD	13	3,452	101,647	13	73,940	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

表 3　ベンチマーク実行結果 (単位は秒)

処理系	Java 実行系	プラットフォーム	tak	ctak	deriv	boyer
本処理系	JBlend	FOMA D901i	86.645	145.539	200.860	10,490.718
本処理系	i アプリ SDK 2.05	Windows PC	4.196	11.050	17.495	460.812
本処理系	JDK 1.5	Windows PC	1.362	5.491	2.367	16.530
JAKLD	同上	同上	2.103	12.000	4.166	29.776
JAKLD	JDK 1.2.1	Solaris EWS	5.957	18.194	10.568	79.760
Skij	同上	同上	9.569	20.828	14.090	906.081
Kawa	同上	同上	1.377	5.738	3.149	19.202

み集めに関する実行時情報を得る手段がないので，この仮説が正しいかどうかを確認することができない．

JDK 1.5 での本処理系と JAKLD とを比較すると，本処理系が約 2 倍の性能であることがわかる．これは，関数呼び出しに JAKLD が reflection を使っているのに対して，本処理系では switch-case を使っているためだろうと考えられる．JDK 1.5 上の JAKLD は，JDK 1.2.1 上の JAKLD の 2.5 倍以上の性能を示しているが，ctak だけは性能が伸びていない．ctak は例外処理を頻繁に使用するもので，JDK の例外処理性能の差が影響したと考えられる．

同じ PC 上でありながら，エミュレータと JDK 1.5 とでは ctak で 2 倍程度，boyer になると 30 倍程度の実行時間の差がある．エミュレータの Java 実行系が KVM であり，JDK とは異なっているが，実行系そのものの性能にさほど大きな相違があるとは考えにくい．エミュレータがどの程度のヒープを使用しているのか知るすべがないのだが，携帯端末のエミュレータであるから，実機に近いサイズに設定してあるだろうと想像できる．つまり，PC 上の本処理系と JAKLD の性能の差は，ヒープサイズの差が原因だと考えるのが妥当である．

KVM/CLDC 上で動作する Lisp 処理系に，MIDPLisp[11] がある．DoJa ではなく，MIDP プロファイルを使用しており，MIDP 準拠の処理系を搭載した端末で動作する．本処理系との主要な相違は，動的束縛 (dynamic binding) を採用していることと，組込み関数ごとにクラスを定義している点であろう．近年の Lisp 処理系は静的束縛 (lexical binding) を採用することが多いが，静的束縛では環境を実現するためにメモリを消費するのが一般的である．スタックを使って動的束縛を実現することにより，メモリの消費を抑えることができる．組込み関数ごとにクラスを定義することにより，組込み関数の呼び出しを Java のメソッド呼び出しに置き換えることができ，本処理系のような switch-case によるディスパッチや，JAKLD で採用した reflection 機能なしでも実装できる．その反面，組込み関数ごとにクラスを用意する必要があり，本格的な処理系を実装すれば，かなりの大きさになることが予想される．現在，MIDPLisp の

サイトで公開されている処理系には，tak ベンチマークを実行するために必要な最低限の組込み関数 (特殊形式を含めても，`car`, `quote`, `+`, `-`, `setq`, `defun`, `>=`, `if`, `cons`, `read-from-string`, `eval` の 11 個) しか定義されていないので，実用的な規模にした場合に処理系がどの程度のサイズになるかは不明である．動的束縛と，クラスによる組込み関数の実現のために，実行性能は本処理系よりも若干優れているようである．MIDPLisp のサイトでベンチマークテストとして使われている (`tak 10 5 0`) を，手元の W-ZERO3 [13](WS004SH，MIDP 対応の JBlend [1]，ヒープサイズ等の詳細は不明) で実行すると，本処理系が 3.075 秒だったのに対して，MIDPLisp は 2.384 秒であった．

評価実験を通して感じたことは，携帯端末の Java 実行系の信頼性が高い点である．boyer の実行にあたっては，他のテスト結果から PC 版の 20 倍程度の実行時間がかかることが分かっていたので，PC 版で 460 秒かかる boyer は，携帯端末では 2.5 時間程度かかることが予想された．最初は，そんなに長時間の実行に耐えられるだろうかと心配であったが，実際には 3 時間近く走り続けて結果を得た．性能評価の際だけではなく，処理系のテスト時点から，Java 実行系に起因するトラブルはいっさい発生していない．本処理系は，携帯端末上ではけっして良い性能を発揮できないが，この信頼性の高さは，おおいに評価できると考えている．

7 おわりに

携帯電話端末，具体的には FOMA 端末で動作する小型の Lisp 処理系を紹介した．本処理系は，実時間ごみ集めに関する研究の副産物として開発された．副産物だけあって，あまり工数をかけずに，一応の動作をする処理系ができあがった．Java を中心とするアプリケーションの開発・実行環境が整っていたからできたことである．

携帯端末の性能，特に CPU 速度とメモリサイズの制約のために，現時点では実行性能がきわめて低い．CPU 速度よりも，メモリが小さいために GC がひんぱんに起きていることが原因だと考えられる．本処理系で高度な計算を行うのは無理があるし，ファンシーなゲームを開発するなら Java で直接記述したほうがよい．本処理系のメリットは，ソースプログラムをコンパイルしないでインタープリタで直接実行できる点であろう．手元の携帯端末を使って，ちょっと簡単な計算をさせる，といった用途が考えられる．現在はまだ開発していないが，DoJa の機能を駆使する API ができれば，携帯端末がユーザに公開しているさまざまな機能を気軽に試すこともできるようになる．

本処理系は JAKLD の精神を受け継ぎ，実行性能よりも処理系コードの可読性を重視している．このために，JAKLD と同じく，処理系の専門家でなくとも，処理系への機能追加が容易に行えるはずである．このことから，学生のプログラミング教育にも利用できないかと考えている．PC をはじめとする通常の計算機上で言語処理系をいろいろと改造することも良い演習にはなるが，普段持ち歩いている携帯端末で自分が改造した処理系が動作するのは，学生にプログラミングに対する興味を与える絶好の機会となるであろう．

ここで報告した処理系は，著者の Web ページでフリーソフトウェアとして公開している [24].

参 考 文 献

[1] アプリックス: JBlend[micro] 概要, http://www.aplix.co.jp/ jp/products/jb_micro/overview.html.

[2] Bothner, P.: Kawa, the Java-based Scheme System, http://www.gnu.org/software/kawa/, 2002.

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

[4] Gabriel, R. P.: *Performance and Evaluation of Lisp System*, MIT Press Series in Computer Science, MIT Press, Cambridge, MA, 1985.

[5] Gosling, J., Joy, B. and Steele Jr., G. L.: *The Java Language Specification*, Addison-Wesley, 1996.

[6] *IEEE Standard for the Scheme Programming Language*, IEEE, 1991.

[7] Ito, T. and Matsui, M.: *A Parallel Lisp Language PaiLisp and Its Kernel Specification*, Lecture Notes in Computer Science 441, Springer-Verlag, 1995, pp. 22–52.

[8] NTT DoCoMo: D901i オンラインマニュアル, http://www.nttdocomo.co.jp/support/manual/online/ mobile/foma/d901i/index.php.

[9] NTT DoCoMo: i アプリコンテンツの概要, http://www.nttdocomo.co.jp/service/imode/make/content/iappli/about/index.html.

[10] Quel プロジェクト: QR コード作成&活用のススメ, qr.quel.jp/, 2004.
[11] 沖ソフトウェア: PHS/携帯電話で動作する Lisp 処理系の作り方, http://www.okisoft.co.jp/esc/go/midplisp.html.
[12] Saiki, H., Konaka, Y., Komiya, T., Yasugi, M. and Yuasa, T.: Real-time GC in JeRTyVM Using the Return-Barrier Method, in *The Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (*ISORC*), 2005, pp. 140–148.
[13] ウィルコム:W-ZERO3(WS004SH), http://www.willcom-inc.com/ja/lineup/ws/004sh/index.html.
[14] Steele Jr., G. L.: *Common Lisp the Language*, Second Edition, Digital Press, 1990.
[15] Sun Microsystems: Connected Limited Device Configuration (CLDC), http://java.sun.com/products/cldc/.
[16] Sun Microsystems: Mobile Information Device Profile (MIDP), http://java.sun.com/products/midp/.
[17] 高野保真, 岩崎英哉: Improving Sequence を第一級の対象とする Scheme コンパイラ, 第 8 回プログラミングおよびプログラミング言語ワークショップ論文集, 2006, pp. 153–162.
[18] Travers, M.: Skij: Scheme in Java for Interactive Debugging and Scripting, http://alphaworks.ibm.com/tech/Skij, 1999.
[19] 山本晃成, 湯淺太一: 末尾再帰の最適化と一級継続を実現するための JVM の機能拡張, 情報処理学会論文誌, Vol. 42 (2001), pp. 37–51.
[20] 湯淺研究室: TUTScheme, http://www.yuasa.kuis.kyoto-u.ac.jp/~komiya/tus_intro.html.
[21] Yuasa, T.: Real-time Garbage Collection on General-purpose Machines, Journal of Systems and Software, Vol. 11, No. 3(1990), pp. 181–198.
[22] 湯淺太一, 中川雄一郎, 小宮常康, 八杉昌宏: リターン・バリア, 情報処理学会論文誌, Vol. 41 (2000), pp. 87–99.
[23] 湯淺太一：Java アプリケーション組込み用 Lisp ドライバ, 情報処理学会論文誌, Vol. 44 (2003), pp. 1–16.
[24] 湯淺太一：Java アプリケーション組込み用の Lisp ドライバ, http://www.yuasa.kuis.kyoto-u.ac.jp/~yuasa/jakld, 2002.
[25] 湯淺太一：Mobile JAKLD ダウンロードページ, http://www.yuasa.kuis.kyoto-u.ac.jp/~yuasa/jakld-i/, 2006