Piper ME

September 20, 2008

Chapter 1

Piper-B for J2ME

1.1 Introduction

Piper-B for J2ME is a port of Yacas/Piper to the J2ME environment and this way make Yacas available on a multitude of mobile devices. The port has the following goals:

1. To do math on a handheld.
2. To have an interesting general purpose programming language available.
3. To gain a better understanding of the current implementation of Yacas.
4. To improve Yacas.
5. To gain enough knowledge about Yacas semantics in order to provide a cleanroom reimplementation, publishable under LGPL.

1.1.1 Platform

The port to Java ME is still work in progress. The targeted plaqtform is Java ME. We use the Java 1.5 memory model, but in a way that it should not impose a restriction on most 1.4 JVM-implementations employed on single-processor machines.
We also make use of generics. By using the *-target jsr14* compiler switch, this should allow us to produce Java ME compatible bytecode. If not, then we might remove the generic type arguments before compiling to Java ME.

1. Immutable Interface for Yacas/Piper numbers
2. Integrating the next-references into LIspObjects. This saves memory. In the long run, list structures should be explicit. An attempt for efficient lists with Yacas semantics has been made in X-Piper's CoreList class.
3. The synchronized java.util.vector has been replaced by unsynchronized structured.

Chapter 2

X-Piper

2.1 X-Piper Core

X-Piper is the experimental reimplementation of Piper-B/Yacas. Also new parts for Piper-ME will be found here. We describe them here, too.

2.1.1 Utility classes

Class for a both-sided extendible array. All operations run in constant time. The class can be used for mutable stacks, queues, and deques with random access.¡br/¿ ¡br/¿ The implementation maintains two arrays, A and B, which are used as round buffers. B has always twice the size of A. The queue is splitted into three sequences: the first part is in B, the second in A, the third again in B exactly opposite to the first part. This scheme allows us to copy no more than one element per update.

```
@SuppressWarnings("unchecked")
public class ExtendibleArray<E> {
```

N is the size of the small array A NTwice is the size of the large array B

```
  int NTwice, N;

  Object A[], B[];
```

The masks for quickly computing the remainders of N and NTwice, respectively

```
  int maskN, maskNTwice;
```

the indizes for maintaining the queues:

```
  int p0, q0, p1, q1, p2, q2;
```

size of the first queue (this is the one in B!)

```
  int b;
```

the initial capacity (must be a power of 2):

```
static final int INITIAL_CAPACITY := 32;
```

debug output of internal structure:

```
public void display() {
  System.out.print("Size:=" + size());
  System.out.print("  p0:=" + p0);
  System.out.print("  q0:=" + q0);
  System.out.print("  p1:=" + p1);
  System.out.print("  p1:=" + q1);
  System.out.print("  p2:=" + p2);
  System.out.print("  q2:=" + q2);
  System.out.println("  b:=" + b);
  System.out.print("Array A:=");
  printSubarray(A);
  System.out.print("Array B:=");
  printSubarray(B);
}

protected void printSubarray(Object[] A) {
  if (A ≠ null) {
    System.out.print("(");
    for (int i := 0; i < A.length; i++) {
      if (i > 0)
        System.out.print(",");
      if (A[i] ≠ null)
        System.out.print(A[i].toString());
      else
        System.out.print(" × ");
    }
    System.out.println(")");
  } else
    System.out.println(" × ");
}

private void init() {
  NTwice := INITIAL_CAPACITY;
  N := NTwice / 2;
  maskN := N - 1;
  maskNTwice := NTwice - 1;
```

if initialCapacity==NTwice then the small array 0 will always be non-existent.

```
  A := null;
  B := new Object[NTwice];
  b := 0;
```

```
    p1 := q1 := p0 := q0 := p2 := q2 := 0;
  }

  public ExtendibleArray() {
    init();
  }

  public ExtendibleArray(int size) {
    init();
    for (int i := 0; i < size; i++)
      addLast(null);
  }

  public final int size() {
    if (NTwice = INITIAL_CAPACITY)
      return b;
    else
      return b + N;
  }

  public final E get(int i) {
    if ((i < 0) || (i ≥ size()))
      throw new java.lang.IndexOutOfBoundsException();
    if ((i < b) || (i ≥ N))
      return (E) B[(i + p0) & maskNTwice];
    else
      return (E) A[(i + p1 - b) & maskN];
  }

  public final E set(int i, E x) {
    if ((i < 0) || (i ≥ size()))
      throw new java.lang.IndexOutOfBoundsException();
    if ((i < b) || (i ≥ N)) {
      int k := (i + p0) & maskNTwice;
      Object old := B[k];
      B[k] := x;
      return (E) old;
    } else {
      int k := (i + p1 - b) & maskN;
      Object old := A[k];
      A[k] := x;
      return (E) old;
    }
  }

  public final void addFirst(E x) {
```

```
    if (NTwice ≠ INITIAL_CAPACITY) {
```

move one item from A to B:

```
      q1--;
      q1 &:= maskN;
      p2--;
      p2 &:= maskNTwice;
      B[p2] := A[q1];
      A[q1] := null;
    }
```

add item at front:

```
    p0--;
    p0 &:= maskNTwice;
    B[p0] := x;

    b++;
    if (p0 = q2)
```

B is full =¿ A is empty We make B to A and create empty B

```
      makeNewB();
  }

  public final void addLast(E x) {
    if (NTwice ≠ INITIAL_CAPACITY) {
```

move one item from A to B

```
      B[q0++] := A[p1];
      A[p1++] := null;
      q0 &:= maskNTwice;
      p1 &:= maskN;
    }
```

add item at tail:

```
    B[q2++] := x;
    q2 &:= maskNTwice;
    b++;

    if (p0 = q2)
```

B is full =¿ A is empty make B to A and create empty B

```
      makeNewB();
  }
```

```
public final boolean add(E x) {
  addLast(x);
  return true;
}

private void makeNewB() {
  N := NTwice;
  NTwice := 2 × NTwice;
  maskN := N − 1;
  maskNTwice := NTwice − 1;
  A := B;
  B := new Object[NTwice];
  b := 0;
  p1 := q1 := p0;
  p0 := q0 := 0;
  p2 := q2 := N;
}

public final E removeFirst() {
  if (size() ≤ 0)
    throw new java.util.NoSuchElementException();
  if (NTwice ≠ INITIAL_CAPACITY) {
    if (b = 0)
```

B is empty =¿ A is full make A to B and create empty A

```
      makeNewA(); //

    if (NTwice ≠ INITIAL_CAPACITY) {
```

move one item from B to A

```
      A[q1++] := B[p2];
      B[p2++] := null;
      q1 &:= maskN;
      p2 &:= maskNTwice;
    }
```

remove item from front

```
    b−−;
    Object x := B[p0];
    B[p0++] := null;
    p0 &:= maskNTwice;
    return (E) x;
  } else if (p0 ≠ q2) {
```

remove item from front

```
      b--;
      Object x := B[p0];
      B[p0++] := null;
      p0 &:= maskNTwice;
      return (E) x;
    } else
      return null;
  }

  public final E removeLast() {
    if (size() ≤ 0)
      throw new java.util.NoSuchElementException();
    if (NTwice ≠ INITIAL_CAPACITY) {
      if (b = 0)
```

B is empty =¿ A is full. make A to B and create empty A:

```
        makeNewA();

      if (NTwice ≠ INITIAL_CAPACITY) {
```

move one item from B to A

```
        p1--;
        p1 &:= maskN;
        q0--;
        q0 &:= maskNTwice;
        A[p1] := B[q0];
        B[q0] := null;
      }
```

remove item from tail

```
      b--;
      q2--;
      q2 &:= maskNTwice;
      Object x := B[q2];
      B[q2] := null;
      return (E) x;
    } else if (p0 ≠ q2) {
```

remove item from tail

```
      b--;
      q2--;
      q2 &:= maskNTwice;
      Object x := B[q2];
      B[q2] := null;
```

```
      return (E) x;
    } else
      return null;
  }

  private void makeNewA() {
```

make A to B and create empty A

```
    NTwice := N;
    N := N / 2;
    maskN := N - 1;
    maskNTwice := NTwice - 1;
    B := A;
    p0 := q2 := p1;
    q0 := p2 := (p1 + N) % NTwice;
    p1 := q1 := 0;

    if (NTwice ≠ INITIAL_CAPACITY) {
      A := new Object[N];
      b := N;
    } else {
      A := null;
      b := NTwice;
    }
  }
```

copy to standard array:

```
  public final Object[] toArray() {
    int n := this.size();
    Object[] A := new Object[n];
    for (int i := 0; i < n; i++)
      A[i] := get(i);
    return A;
  }

  public void insert(int i, E x) {
    if (i = size()) {
      add(x);
      return;
    }
    int n := size()-1;
    Object tmp := get(n);
    addLast((E)tmp);
    for (int j := n; j > i; j--)
      set(j,get(j-1));
    set(i,x);
```

```
  }

}// class
```

Wrapper for Extendinbe Array in order to provide a mutable stack.

```
public final class RandomAccessStack<E> extends ExtendibleArray<E>
{
    public E pop() { return removeFirst(); }
    public E top() { return get(0); }
    public void push(E item) { addFirst(item);}
}
```

2.1.2 The Core List Class

This is a list class with Yacas semantics of lists. That means, mutual sharing of sublists is possible, and destructive updates to those sublists will affect all participating lists. This implementation is optimized forcomputing with vectors. Lists obtained from an array will retain the array structure unless tailing occurs. Once the tail of a list is obtained, the array structure will be rearranged in such way that mutual sharing is possible. That implies t split off the list's first element from the array. As a result, after tailing the entire list, random access costs linear time instead of constant time. This is the same as in the original Yacas list implementation. The purpose of this class is to provide constant time random access for vector processing routines that keep their vector data local, while at the same time stick with Yacas' semantics that arrays are lists. An attempt is made to provide thread safety to a certain degree. The exact meaning is that the restructuring of the array remains invisible to the user, while the user remains responsible for the visibility of destructive updates, in particular happened-before initialization of elements. This class works best if used without destructive updates and without tailing, since only volatile accesses to the array reference is needed. In many cases, the Java-VM can optimize away synchronization, since in this case the array reference is never mutated after initialization.

1. Generate unit tests.

2. Add functions.

```
public final class CoreList {

  private Object value;
  private volatile Object rest;
```

If the rest is null, the list is empty. Otherwise the list's first value is stored in the value-attribute. The rest of the list is either a reference to a sublist of the same type, or a reference to an array c of $n+1$ objects. In the latter case, $c[0]$ is a reference to a sublist and $c[1..n]$ are list elements. The complete list is then (in order) the value-attribute plus $c[1..n]$ concatenated with the sublist $c[0]$.

```
public static final CoreList emptyList := new CoreList(null);
```

Constructor for list with exactly one element:

```
private CoreList(Object v) {
  value := v;
  rest := emptyList;
}
```

Constructor for appending one element at front of a list:

```
private CoreList(Object v, CoreList rest) {
  value := v;
  if (rest = null)
    this.rest := emptyList;
  else
    this.rest := rest;
}
```

Constructor for appending a subarray at front of a list:

```
private CoreList(Object[] arr, int pos, int length , CoreList rest) {
  value := arr[pos];
  if (rest = null)
    rest := emptyList;
  if (length > 1) {
    Object[] chunk := new Object[length];
    this.rest := chunk;
    chunk[0] := rest;
    int j := 1;
    for (int i := pos+1; i < pos + length; i++) {
      chunk[j++] := arr[i];
    }//for
  } else {
      this.rest := rest;
  }//else
}
```

test for empty list

```
public boolean isEmpty() {
  return this = emptyList;
 }

public CoreList push(Object a) {
  return new CoreList(a, this);
}
```

```
public CoreList push(Object[] a, int pos, int length) {
  if (length = 0)
    return this;

  return new CoreList(a, pos, length, this);
}
```

returns the tail of the list. If necessary, the list is restructured so that modifications of the sublists will be shared. Note that subsequent tailing of the list leads to a degenerated random access time for the original(!) list. Lists that are used as vectors should never be tailed, and not be exported to code that does.

```
public CoreList tail() {
  if (rest.getClass() = CoreList.class)
    return (CoreList) rest;
  else {
    splitFirstFromTail();
    return (CoreList) rest;
  }//else
}
```

result is the same as tail().tail()... /*n*times), except that only the affected parts of the list are restructured.

TODO

```
public CoreList tail(int n) {
  throw new UnsupportedOperationException();
}
```

The split method is synchronized. This reflects the user's ecpectation that read-only accesses (like tailing) should not need user-level synchronization during absence of writes.

```
private synchronized void splitFirstFromTail() {
  Object restLocalCopy := this.rest;
```

Note that it it is not necessary to obtain a local copy of the reference to rest of the list. The method is synchronized and no unsynchronized method modifies this refernce in the current implementation. We do, however, obtain a local copy to be prepared if we find out that synchronizing read access is up to the user and we drop the method's synchronized modifier in favor of efficiency.

If he chunk points to a proper list, nothing needs to be done.

```
  if (restLocalCopy.getClass() = CoreList.class) {
    return;
  }//if
  Object[] chunk := (Object[]) restLocalCopy;
```

We use logarithmic splitting in order to obtain amortized constant time for obtaining the rest of a list.

```
    int N := chunk.length - 1;
    int m := 1;
    while (m × 2 ≤ N) {
      m := 2 × m;
    }//while
    CoreList rest := (CoreList) chunk[0];
    while (m > 1) {
    rest := new CoreList(chunk, m, N-m+1, (CoreList) rest);
    N := m-1;
    m := m / 2;
    }//while
    this.rest := new CoreList(chunk[1],rest);
  }
```

Prints the internal structure into a string

```
  public String debug() {
    CoreList list := this;
    StringBuffer s := new StringBuffer();
    s.append('(');
    while (!list.isEmpty()) {
      s.append(list.value.toString());
      Object restLocalCopy := list.rest;
      if (restLocalCopy.getClass() = CoreList.class)
        list := (CoreList) restLocalCopy;
      else {
        Object[] chunk := (Object[]) restLocalCopy;
        list := (CoreList) chunk[0];
        for (int i := 1; i < chunk.length; i++) {
          s.append(' ');
          s.append(chunk[i].toString());
        }//for
      }//else
      if (restLocalCopy ≠ null)
        s.append(")->(");
    }//while
    s.append(')');
    return s.toString();
  }

  public String toString() {
    CoreList list := this;
    StringBuffer s := new StringBuffer();
    s.append('(');
```

```
    while (!list.isEmpty()) {
      s.append(list.value.toString());
      Object restLocalCopy := list.rest;
      if (restLocalCopy.getClass() = CoreList.class)
        list := (CoreList) restLocalCopy;
      else {
        Object[] chunk := (Object[]) restLocalCopy;
        list := (CoreList) chunk[0];
        for (int i := 1; i < chunk.length; i++) {
          s.append(' ');
          s.append(chunk[i].toString());
        }//for
      }//else
      if (!list.isEmpty())
        s.append(' ');
    }//while
    s.append(')');
    return s.toString();
  }
```

Get the value at position i.

```
  public Object get(int i) {
    CoreList list := this;
    while (!list.isEmpty()) {
      if (i = 0) return list.value;
      Object restLocalCopy := list.rest;
      if (restLocalCopy.getClass() = CoreList.class) {
        list := (CoreList) restLocalCopy;
        i := i - 1;
      } else {
        Object[] chunk := (Object[]) restLocalCopy;
        list := (CoreList) chunk[0];
        if(i < chunk.length) {
          return chunk[i];
        }
        i := i - chunk.length;
      }//else
    }//while
    throw new IndexOutOfBoundsException();
  }
```

Destructively set element at position i to value e. The old value ist returned.

```
  public Object set(int i, Object e) {
    CoreList list := this;
    while (!list.isEmpty()) {
      if (i = 0) {
```

```
        Object x := list.value;
        list.value := e;
        return x;
      }//if
      Object restLocalCopy := list.rest;
      if (restLocalCopy.getClass() = CoreList.class) {
        list := (CoreList) restLocalCopy;
        i := i - 1;
      } else {
        Object[] chunk := (Object[]) restLocalCopy;
        list := (CoreList) chunk[0];
        if(i < chunk.length) {
          Object x := chunk[i];
          chunk[i] := e;
          return x;
        }//if
        i := i - chunk.length;
      }//else
    }//while
    throw new IndexOutOfBoundsException();
  }

  public static void main(String[] args) {
    int N := 10;
    Integer[] arr := new Integer[N];
    for (int i := 0; i < N; i++) {
      arr[i] := new Integer(i);
    }//for
    CoreList c := new CoreList(arr, 0, N, emptyList);
    System.out.println(c.debug());
    for (int i := 0; i < N; i++) {
      c.set(i, new Integer(((Integer)c.get(i)).intValue()+100));
      System.out.println(c);
    }//for
    System.out.println(c.debug());
    CoreList d := c;
    while (!d.isEmpty()) {
      d.set(0, new Integer(((Integer)d.get(0)).intValue()+100));
      System.out.println(c);
      d := d.tail();
    }//while
    System.out.println(c.debug());
  }
}//class
```

2.2 Basic Arithmetic

Radix-2 FFT. This class is intended for fast multiplication of very large numbers.
TODO: error handling

```
public class Fourier {
```

convolution that runs in $O(n^2)$ time. Use this function for small arrays or in order to test fastConvolute(...)

```
  public static double[] convolute(double[] F1, double F2[]) {
    int k1 := F1.length - 1;
    int k2 := F2.length - 1;
    int n := k1 + k2 + 1;
    double[] F := new double[n];
    for (int i := 0; i < n; i++)
      F[i] := 0.0;
    for (int i := 0; i ≤ k1; i++)
      for (int j := 0; j ≤ k2; j++)
        F[i + j] +:= F1[i] × F2[j];
    return F;
  }
```

Fast convolution that runs in $O(nlogn)$ time using FFT. The current implementation copies a lot of arrays and might be a bit inefficient but therefore easier to debug.

```
  public static double[] fastConvolute(double[] F1, double F2[]) {
    int k1 := F1.length;
    int k2 := F2.length;
    if (k1 × k2 ≤ 512 × 512)
      return convolute(F1, F2);
    int n := k1 + k2 - 1;
    double[] F := new double[n];
    int K := 1;
    while (K < n)
      K <≤ 1;
    double aRe[] := new double[K];
    double aIm[] := new double[K];
    double bRe[] := new double[K];
    double bIm[] := new double[K];
    for (int i := 0; i < K; i++)
      aIm[i] := bIm[i] := 0.0;
    for (int i := 0; i < k1; i++)
      aRe[i] := F1[i];
    for (int i := k1; i < K; i++)
      aRe[i] := 0.0;
    for (int i := 0; i < k2; i++)
      bRe[i] := F2[i];
```

```
    for (int i := k2; i < K; i++)
      bRe[i] := 0.0;
    computeFFT(aRe, aIm, aRe, aIm);
    computeFFT(bRe, bIm, bRe, bIm);
    for (int i := 0; i < K; i++) {
      double _aRe := aRe[i];
      double _aIm := aIm[i];
      double _bRe := bRe[i];
      double _bIm := bIm[i];
      aRe[i] := _aRe  ×  _bRe - _aIm  ×  _bIm;
      aIm[i] := _aRe  ×  _bIm + _aIm  ×  _bRe;
    }
    ;
    computeIFFT(aRe, aIm, aRe, aIm);
    for (int i := 0; i < n; i++)
      F[i] := aRe[i];

    return F;
  }
```

several bit and index manipulation routines

```
  private static int log2(int K) {
    int k := 0;
    int N := K;
    int S := 1;
    while (N ≠ 1) {
      S <≤ 1;
      N >≥ 1;
      k +:= 1;
    }
    if (S ≠ K)
      System.out.println("K is not a power of 2.");
    return k;

  }

  private static int reverseBits(int i, int k) {
    int j := 0;
    for (; k > 0; k--) {
      j <≤ 1;
      j |:= i & 1;
      i >≥ 1;
    }
    return j;
  }
```

```
private static void reverseBitsOfIndex(double[] src, double[] dst) {
  int K := src.length;
  int k := log2(K);
  boolean tmp := (src = dst);
  if (tmp)
    dst := new double[K];

  for (int i := 0; i < K; i++) {
    int j := reverseBits(i, k);
    dst[i] := src[j];
  }

  if (tmp)
    for (int i := 0; i < K; i++)
      src[i] := dst[i];
}
```

compute 2^n roots on the circle, the arrays must have a size of 2^n

```
public static void computeFourierCoeff(double[] wRe, double[] wIm) {
  int K := wRe.length;
  if (K ≠ wIm.length)
    System.out.println("wRe and wIm have different length.");
  if (K = 0)
    return;
  int k := log2(K);
  wRe[0] := 1.0;
  wIm[0] := 0.0;
  if (K = 1)
    return;
  wRe[1] := -1.0;
  wIm[1] := 0.0;
  if (K = 2)
    return;
  wRe[2] := 0.0;
  wIm[2] := 1.0;
  wRe[3] := 0.0;
  wIm[3] := -1.0;
  int j := 4;
  int c := 2;
  for (int r := 2; r < k; r++) {
    double pRe := Math.sqrt((1 + wRe[c]) / 2);
    double pIm := wIm[c] / (2 × pRe);
    wRe[j] := pRe;
    wIm[j] := pIm;
    c := j;
    j++;
```

```
      for (int l := 1; l < c; l++) {
        wRe[j] := wRe[l] × pRe − wIm[l] × pIm;
        wIm[j] := wRe[l] × pIm + wIm[l] × pRe;
        j++;
      }
    }
  }
```

compute FT in $O(nlogn)$ time:

```
  public static void computeFFT(double[] aRe, double[] aIm,
      double[] bRe, double[] bIm) {
    int K := aRe.length;
    if ((bRe.length ≠ K) || (bIm.length ≠ K) || (aIm.length ≠ K))
      System.out.println("Arrays for FFT have different lengths.");

    double[] uRe := new double[K];
    double[] uIm := new double[K];
    for (int i := 0; i < K; i++) {
      uRe[i] := aRe[i];
      uIm[i] := aIm[i];
    }
    double[] wRe := new double[K];
    double[] wIm := new double[K];
    computeFourierCoeff(wRe, wIm);
    int T := K >> 1;
    int F := 1;
    boolean stop := false;

    while (!stop) {
      for (int s := 0; s < F; s++) {
        int s0Offs := s × 2 × T;
        int s1Offs := s0Offs + T;
        double w_s0Re := wRe[2 × s];
        double w_s0Im := wIm[2 × s];
        for (int t := 0; t < T; t++) {
          int s0t := s0Offs + t;
          int s1t := s1Offs + t;
          double u_s0tRe := uRe[s0t];
          double u_s0tIm := uIm[s0t];
          double u_s1tRe := uRe[s1t];
          double u_s1tIm := uIm[s1t];
          double pRe := u_s1tRe × w_s0Re − u_s1tIm × w_s0Im;
          double pIm := u_s1tRe × w_s0Im + u_s1tIm × w_s0Re;
          uRe[s0t] := u_s0tRe + pRe;
          uIm[s0t] := u_s0tIm + pIm;
          uRe[s1t] := u_s0tRe − pRe;
```

```
          uIm[s1t] := u_s0tIm - pIm;
        }
      }
      if (T = 1)
        stop := true;
      else
        T >≥ 1;
      F <≤ 1;
    }
    reverseBitsOfIndex(uRe, bRe);
    reverseBitsOfIndex(uIm, bIm);
  }
```

compute inverse FT in $O(nlogn)$ time using FFT

```
  public static void computeIFFT(double[] aRe, double[] aIm,
      double[] bRe, double[] bIm) {
    int K := aRe.length;
    computeFFT(aRe, aIm, bRe, bIm);
    double r := 1 / (double) K;
    for (int i := 0; i < K; i++) {
      bRe[i]  × := r;
      bIm[i]  × := r;
    }
    int i := 1;
    int j := K - 1;
    while (i < j) {
      double tmp := bRe[i];
      bRe[i] := bRe[j];
      bRe[j] := tmp;
      tmp := bIm[i];
      bIm[i] := bIm[j];
      bIm[j] := tmp;
      i++;
      j--;
    }
  }
}// class
```