McMaster University
Department of Computing and Software
Dr. Wolfram Kahl

SE 3E03, Fall 2002
Assignment 2
Solution Hints

Design and Selection of Programming Languages

5th September 2002

Your First Cup of **Java**

Problem 7 (First Cup of Java)

Follow the Java tutorial at the following web site:

URL: http://java.sun.com/docs/books/tutorial/

On the department's SUN computers, the JDK is installed, and you should start with completing the "Your First Cup of Java" trail. (If you haven't been using Emacs or XEmacs as your editor so far, this is a good opportunity to get started.)

Then, follow the whole "Getting Started" trail and complete the exercises on your own before proceeding to the answers page. While reading the "The Java Phenomenon" lesson, make notes of any critical questions that you would like to ask.

Finally, from the trail "Learning the Java Language", follow the "Language Basics" lesson, and again complete all the exercises.

Problem 8 (Java versus Oberon and C)

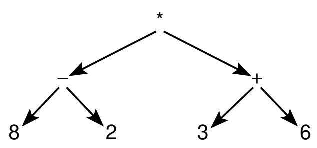
Identify differences between Java, Oberon-2, and C as you notice them.

In particular, try to find out where just the terminology is different, and where the concepts or restrictions are different.

Enter your observations into the table on the next page, and also complete the given entries. As last resort, you may refer to the Java 2 Language Specification — there are pointers on the course page.

Problem 9 (Data Stuctures Review)

Aritmetic expressions can be understood as tree structures; the tree corresponding to $(8-2)*(3+6)$ is



a) Provide Oberon-2 data type definitions for such arithmetic expression trees.

b) Provide an Oberon-2 function definition for an evaluation function for such arithmetic expression trees.

Remember your Software Engineering principles!

Solution: For now, leaves may be just integers. Variables are good, but will be demanded only in a follow-up problem.

From an object-oriented point of view, the most important kinds of things we are dealing with are arithmetic expressions, so we should have a class `Expr` for these.

From the example, we can identify two subclasses: numbers, and expressions built from two subexpressions via application of a binary operator.

The previous sentence mentions two other kinds of entities we are dealing with: numbers, and binary operators. For numbers, we have to decide which kind of number to use — `LONGINT` or `LONGREAL` are reasonable default choices (however, `obc` does not implement `LONGINT`).

For binary operators, we could define a separate class, or we might use their textual representation; in the latter case, we face the choice whether to use characters or strings.

Because of Oberon's "special" character syntax, we introduce a few constants:

Oberon Expr Chars[7] ≡
```
{CONST
     times = 2AX;
     plus = 2BX;
     minus = 2DX;
 }
```

Since we cannot find anything that all three expression classes have in common, the base class is going to be empty. In addition, since every expression belongs to one of the subclasses, the base class should be abstract — no objects of this class are needed. However, since we need the procedure as a member of the base class, with `obc` there does not seem to be a way to omit its body.

In Oberon, the question about access methods may be postponed, so we can start with the type definitions:

Oberon Expr Type[8] ≡
```
{TYPE Num* = INTEGER;
      Expr* = POINTER TO ExprRec;
      ExprRec = RECORD END;
      NumberExpr = POINTER TO NumberExprRec;
      NumberExprRec = RECORD(ExprRec) number : Num END;
      BinOpExpr = POINTER TO BinOpExprRec;
      BinOpExprRec = RECORD(ExprRec) op : CHAR;
                                     left, right: Expr END;
 }
```

The one kind of access specified so far is evaluation; this has to be a type-directed procedure (class method). Although we only export the base class, the different type-directed instances of this method all need to have the same export status.

Oberon Expr Eval[9] ≡
```
{PROCEDURE (e: Expr) eval* () : Num;
 BEGIN
   RETURN 0;
 END eval;
 PROCEDURE (e : NumberExpr) eval* () : Num;
 BEGIN
     RETURN e.number;
 END eval;
 PROCEDURE (e : BinOpExpr) eval* () : Num;
 VAR
   l,r : Num;
 BEGIN
   l := e.left.eval();
   r := e.right.eval();
```

```
    CASE e.op OF
      plus  : RETURN l+r
    | times : RETURN l*r
    | minus : RETURN l-r
    END;
  END eval;
  }
```

Although not mentioned in the problem statement, it still makes sense to export constructors for our expression data type:

Oberon Expr Constructors[10] ≡

```
 {PROCEDURE numExpr* (n : Num) : Expr;
  VAR
      ne : NumberExpr;
  BEGIN
    NEW(ne);
    ne.number := n;
    RETURN ne
  END numExpr;

  PROCEDURE binOpExpr* (o : CHAR; l,r : Expr) : Expr;
  VAR
      b : BinOpExpr;
  BEGIN
    NEW(b);
    b.op := o;
    b.left := l;
    b.right := r;
    RETURN b
  END binOpExpr;
  }
```

We use these to build the example expression:

Oberon Expr Example[11] ≡

```
 {PROCEDURE exampleExpr* () : Expr;
  BEGIN
    RETURN binOpExpr(times, binOpExpr(minus, numExpr(8), numExpr(2))
                          , binOpExpr(plus , numExpr(3), numExpr(6)) );
  END exampleExpr;
  }
```

The main program just evaluates the example expression:

Expr.m[12] ≡

```
 {MODULE Expr;

  IMPORT Out;
  Oberon Expr Chars[7]
  Oberon Expr Type[8]
  Oberon Expr Constructors[10]
  Oberon Expr Eval[9]
  Oberon Expr Example[11]
```

```
VAR
  e : Expr;
BEGIN
  e := exampleExpr();
  Out.Int(e.eval(),0); Out.Ln;
END Expr.
}
```

This macro is attached to an output file.