Computer Science 3 - 2008

Programming Language Translation

Practical for Weeks 25 - 26, beginning 13 October 2008 - solutions

Sources of full solutions for these problems may be found on the course web page as the file `PRAC25A.ZIP` (Java) or `PRAC25AC.ZIP` (C#).

Task 2 - Better use of the debugging pragma

The extra pragmas needed in the refined Parva compiler are easily introduced. We need some static fields:

```
public static boolean
  debug = false,
  listCode = false,
  warnings = true;
```

The definitions of the pragmas are done in terms of these:

```
PRAGMAS
  DebugOn     = "$D+" .         (. debug = true; .)
  DebugOff    = "$D-" .         (. debug = false; .)
  WarnOn      = "$W+" .         (. warnings = true; .)
  WarnOff     = "$W-" .         (. warnings = false; .)
  CodeOn      = "$C+" .         (. listCode = true; .)
  CodeOff     = "$C-" .         (. listCode = false; .)
```

It is convenient to be able to set the options with command line parameters as well. This involves a straightforward change to the `Parva.frame` file:

```
    for (int i = 0; i < args.length; i++) {
      if (args[i].toLowerCase().equals("-l")) mergeErrors = true;
      else if (args[i].toLowerCase().equals("-d")) Parser.debug = true;
**    else if (args[i].toLowerCase().equals("-w")) Parser.warnings = false;
**    else if (args[i].toLowerCase().equals("-c")) Parser.listCode = true;
      else inputName = args[i];
    }
    if (inputName == null) {
      System.err.println("No input file specified");
      System.err.println("Usage: Parva [-l] [-d] [-w] [-c] source.pav [-l] [-d] [-w] [-c] ");
      System.err.println("-l directs source listing to listing.txt");
      System.err.println("-d turns on debug mode");
**    System.err.println("-w suppresses warnings");
**    System.err.println("-c lists object code (.cod file)");
**    System.exit(1);
    }
```

Finally, the following change to the frame file gives the option of suppressing the generation of the `.COD` listing.

```
if (Parser.listCode) PVM.listCode(codeName, codeLength);
```

Task 3 - Learning many languages is sometimes confusing

To be as sympathetic as possible in the face of confusion between various operators is easily handled - we make the parsers that identify these operators accept the incorrect ones, at the expense of generating an error message (or, if you want to be really kind, issue a warning only):

```
    EqualOp<out int op>                 (. op = CodeGen.nop; .)
    =    "=="                           (. op = CodeGen.ceq; .)
       | "!="                           (. op = CodeGen.cne; .)
**     | "="                            (. SemError("== intended?"); .)
**     | "<>"                           (. SemError("!= intended?"); .) .

    AssignOp
**  =    "="
**     | ":="                           (. SemError("= intended?"); .) .
```

Similarly, recovering from the spurious intoduction of `then` into an *IfStatement* is quite easily achieved:

```
    IfStatement<StackFrame frame>         (. Label falseLabel = new Label(!known); .)
    =  "if" "(" Condition ")"             (. CodeGen.branchFalse(falseLabel); .)
**     [ "then"                           (. SemError("then is not used in Parva"); .)
       ] Statement<frame>                 (. falselabel.here(); .) .
```

Task 4 - Suppressing some error messages

The suggestion was made that when an identifier was not found in the symbol table, a suitable entry could quietly be inserted into the table in the hope of reducing the number of irritating "undeclared identifier" messages that might otherwise pop up. This is quite easily done from within the production for *Designator*. Note the way in which we modify the newly inserted `entry` if we establish that the undeclared identifier appears to be of a reference type. Care must be taken not to mess up the original sentinel node; note how `entry` is reallocated.

```
    Designator<out DesType des>           (. String name;
                                             int indexType; .)
    =  Ident<out name>                    (. Entry entry = Table.find(name);
**                                           boolean notDeclared = !entry.declared;
**                                           if (notDeclared) {
**                                             SemError("undeclared identifier");
**                                             entry = new Entry(); // new is critical
**                                             entry.name = name;
**                                             entry.kind = Entry.Var;
**                                             entry.type = Entry.noType;
**                                             entry.offset = 0;
**                                             Table.insert(entry);
**                                           }
                                             des = new DesType(entry);
                                             if (entry.kind == Entry.Var) CodeGen.loadAddress(entry); .)
**     [  "["                             (. if (notDeclared) entry.type++;
**                                           else if (isRef(des.type)) des.type--;
                                             else SemError("unexpected subscript");
                                             if (entry.kind != Entry.Var)
                                               SemError("unexpected subscript");
                                             CodeGen.dereference(); .)
              Expression<out indexType>   (. if (!isArith(indexType)) SemError("invalid subscript type");
                                             CodeGen.index(); .)
          "]"
       ] .
```

Task 5 - Repeat these exercises until you get the idea

Adding the basic *Repeat* loop to Parva is very easy, since all that is needed is a "backward" branch.

```
**  RepeatStatement<StackFrame frame>     (. Label startLoop = new Label(known); .)
**  =  "repeat" { Statement<frame>  }
**     "until" "(" Condition ")" WEAK ";" (. CodeGen.branchFalse(startLoop); .) .
```

Task 6 - You had better do this one or else….

Adding the optional, possibly repeated, *elsif* clauses requires a little care to make sure that the correct sequence of branches is generated. It is possible, of course, that the `outLabel` will turn out never to have been "used", but fortunately the `here` method of the `Label` class takes this in its stride!

```
    IfStatement<StackFrame frame>         (. Label falseLabel = new Label(!known);
**                                           Label outLabel = new Label(!known); .)
    =  "if" "(" Condition ")"             (. CodeGen.branchFalse(falseLabel); .)
       [ "then"                           (. SemError("then is not used in Parva"); .)
       ] Statement<frame>
**     {                                  (. CodeGen.branch(outLabel);
**                                           falseLabel.here();
**                                           falseLabel = new Label(!known); .)
**       "elsif" "(" Condition ")"        (. CodeGen.branchFalse(falseLabel); .)
**       [ "then"                         (. SemError("then is not used in Parva"); .)
**       ] Statement<frame>
**     }
**     (   "else"                         (. CodeGen.branch(outLabel);
**                                           falseLabel.here(); .)
**         Statement<frame>
**       | /* no else part */             (. falseLabel.here(); .)
**     )                                  (. outLabel.here(); .) .
```

Most submissions, however, were on the lines of the production below. This "works", but can generate `BRN` instructions where none are needed. For example, source code like

```
if (i == 12) k = 56;
```

leads to object code like

```
12    LDA  0
14    LDV
15    LDC  12
17    CEQ
18    BZE  27
20    LDA  5
22    LDC  56
24    STO
25    BRN  27        // unnecessary
27    ....
```

```
    IfStatement<StackFrame frame>         (. Label falseLabel = new Label(!known);
                                             Label outLabel = new Label(!known); .)
    =  "if" "(" Condition ")"             (. CodeGen.branchFalse(falseLabel); .)
       [ "then"                           (. SemError("then is not used in Parva"); .)
**     ] Statement<frame>                 (. CodeGen.branch(outLabel);
                                             falseLabel.here(); .)
       { "elsif" "(" Condition ")"        (. falseLabel = new Label(!known);
                                             CodeGen.branchFalse(falseLabel); .)
         [ "then"                         (. SemError("then is not used in Parva"); .)
**       ] Statement<frame>               (. CodeGen.branch(outLabel);
                                             falseLabel.here(); .)
       }
       [ "else"  Statement<frame>  ]      (. outLabel.here(); .) .
```

Task 7 - This has gone on long enough - time for a break

The syntax of the *BreakStatement* is, of course, trivial. The catch is that one has to allow these statements only in the context of loops. To find a context-free grammar with this restriction is not worth the effort. As with nested comments in languages that allow them, it is easier just to have a (global) counter that is incremented and decremented as parsing of loops starts and finishes.

However, loops must be handled in a way that allows them to be nested, with all the breaks in each loop directed at the correct place for that loop - and many of these involve forward references. As it happens, the *Label* class we already use allows for this to be handled neatly, and we can get away with using a global label. However, we need a little local stack to be introduced in each loop parsing production, so that this global label can be kept up to date.

Once you have seen the solution it probably looks almost obvious. One way is as follows: Extra static fields are declared in the parser (declared at the top of the ATG file):

```
**  static int loopLevel = 0;                    // = 0 outside of loops, > 0 inside loops
**  static Label loopExit = new Label(!known);   // current target for "break" statements
```

and the production for the *BreakStatement* then follows as

```
    BreakStatement
**  =  "break"                            (. if (loopLevel == 0)
**                                             SemError("break is not within a loop");
**                                          CodeGen.branch(loopExit); .)
**     WEAK ";" .
```

The *WhileStatement* and *RepeatStatement* productions now have quite a lot of extra actions:

```
**  WhileStatement<StackFrame frame>      (. loopLevel++;
**                                          Label oldExit = loopExit;
**                                          loopExit = new Label(!known);
                                            Label startLoop = new Label(known); .)
    =  "while" "(" Condition ")"          (. CodeGen.branchFalse(loopExit); .)
       Statement<frame>                   (. CodeGen.branch(startLoop);
**                                          loopExit.here();
**                                          loopExit = oldExit;
**                                          loopLevel--; .) .
```

```
**  RepeatStatement<StackFrame frame>     (. loopLevel++;
**                                           Label oldExit = loopExit;
**                                           loopExit = new Label(!known);
                                             Label startLoop = new Label(known); .)
    =  "repeat" { Statement<frame>  }
       WEAK "until"
       "(" Condition ")" WEAK ";"         (. CodeGen.branchFalse(startLoop);
**                                           loopExit.here();
**                                           loopExit = oldExit;
**                                           loopLevel--; .) .
```

Another solution, which some might think of, dispenses with the counter by initializing `loopExit` to `null`:

```
**  static Label loopExit = null;                 // current target for "break" statements
```

and the production for the *BreakStatement* follows as

```
    BreakStatement
**  =  "break"                            (. if (loopExit == null)
**                                             SemError("break is not within a loop");
**                                           else CodeGen.branch(loopExit); .)
**     WEAK ";" .
```

And, for example the *RepeatStatement* production now becomes:

```
**  RepeatStatement<StackFrame frame>     (. Label oldExit = loopExit;
**                                           loopExit = new Label(!known);
                                             Label startLoop = new Label(known); .)
    =  "repeat" { Statement<frame>  }
       WEAK "until"
       "(" Condition ")" WEAK ";"         (. CodeGen.branchFalse(startLoop);
**                                           loopExit.here();
**                                           loopExit = oldExit; .)
```

Task 8 - Break over, let's continue

The modifications needed to incorporate a *ContinueStatement* are similar to those already seen for the *BreakStatement*, and can be handled with a similar save and restore of a `loopContinue` label:

```
**  ContinueStatement
**  =  "continue"                         (. if (loopLevel == 0)
**                                             SemError("continue is not within a loop");
**                                           CodeGen.branch(loopContinue); .)
**     WEAK ";" .

    WhileStatement<StackFrame frame>      (. loopLevel++;
**                                           Label oldContinue = loopContinue;
                                             Label oldExit = loopExit;
                                             loopExit = new Label(!known);
**                                           loopContinue = new Label(known); .)
    =  "while" "(" Condition ")"          (. CodeGen.branchFalse(loopExit); .)
**     Statement<frame>                   (. CodeGen.branch(loopContinue);
                                             loopExit.here();
                                             loopExit = oldExit;
**                                           loopContinue = oldContinue;
                                             loopLevel--; .) .

    RepeatStatement<StackFrame frame>     (. loopLevel++;
**                                           Label oldContinue = loopContinue;
                                             Label oldExit = loopExit;
**                                           loopContinue = new Label(!known);
                                             Label startLoop = new Label(known);
                                             loopExit = new Label(!known); .)
    =  "repeat"
         { Statement<frame>  }
**     WEAK "until"                       (. loopContinue.here(); .)
       "(" Condition ")" WEAK ";"         (. CodeGen.branchFalse(startLoop);
                                             loopExit.here();
                                             loopExit = oldExit;
**                                           loopContinue = oldContinue;
                                             loopLevel--; .) .
```

A notable feature of most submissions was that their authors were clearly confused as to where a *ContinueStatement* should branch, so study these solutions carefully!

Task 9 - Make the change; enjoy life; upgrade now to Parva++ (Ta-ra!)

It might not at first have been obvious, but hopefully everyone eventually saw that this extension is handled by clever modifications to the *Assignment* production, which has to be factorized in such a way as to avoid LL(1) conflicts. The code below achieves all this (including the tests for compatibility that some students probably omitted) by assuming the existence of a few new machine opcodes, as suggested in the textbook.

```
        Assignment                          (. int expType;
                                               DesType des; .)
        =  Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                                 SemError("invalid assignment"); .)
**      (  AssignOp
**         Expression<out expType>          (. if (!assignable(des.type, expType))
**                                                 SemError("incompatible types in assignment");
**                                            CodeGen.assign(des.type); .)
**        | "++"                            (. if (!isArith(des.type))
**                                                 SemError("arithmetic destination needed");
**                                            CodeGen.increment(des.type); .)
**        | "--"                            (. if (!isArith(des.type))
**                                                 SemError("arithmetic destination needed");
**                                            CodeGen.decrement(des.type); .)
**      )
        WEAK ";" .
```

while the prefix forms of the statements are somewhat easier:

```
        IncOrDecStatement                   (. DesType des;
                                               boolean inc = true; .)
        =  ( "++" | "--"                    (. inc = false; .)
           ) Designator<out des>            (. if (des.entry.kind != Entry.Var)
                                                 SemError("invalid assignment");
                                            if (!isArith(des.type))
                                                 SemError("arithmetic destination required");
                                            if (inc) CodeGen.increment(des.type);
                                            else CodeGen.decrement(des.type); .)
           WEAK ";" .
```

The extra code generation routines are straightforward:

```
public static void increment(int type) {
// Generates code to increment the value found at the address currently
// stored at the top of the stack.
  emit(PVM.inc);
}

public static void decrement(int type) {
// Generates code to decrement the value found at the address currently
// stored at the top of the stack.
  emit(PVM.dec);
}
```

As usual, the extra opcodes in the PVM make all this easy to achieve at run time. Some submissions might have forgotten to include the check that the address was "in bounds". I suppose one could argue that if the source program were correct, then the addresses could not go out of bounds, but if the interpreter were to be used in conjunction with a rather less fussy assembler (as we had in earlier practicals) it would make sense to be cautious.

```
case PVM.inc:           // int ++
  adr = pop();
  if (inBounds(adr)) mem[adr]++;
  break;

case PVM.dec:           // int --
  adr = pop();
  if (inBounds(adr)) mem[adr]--;
  break;
```

Task 10 - Let's operate like C

This was the biggest "hack" in this practical, but was hopefully straightforward, since you had been given the unattributed grammar in which the required operator precedence levels had already been sorted out for you. The code follows - note how the short-circuit semantics are achieved for the Boolean operators:

```
Expression<out int type>              (. int type2;
                                         Label shortcircuit = new Label(!known); .)
=  AndExp<out type>
   { "||"                             (. CodeGen.booleanOp(shortcircuit, CodeGen.or); .)
     AndExp<out type2>                (. if (!isBool(type) || !isBool(type2))
                                            SemError("Boolean operands needed");
                                         type = Entry.boolType; .)
   }                                  (. shortcircuit.here(); .)
.

AndExp<out int type>                  (. int type2;
                                         Label shortcircuit = new Label(!known); .)
=  EqlExp<out type>
   { "&&"                             (. CodeGen.booleanOp(shortcircuit, CodeGen.and); .)
     EqlExp<out type2>                (. if (!isBool(type) || !isBool(type2))
                                            SemError("Boolean operands needed");
                                         type = Entry.boolType; .)
   }                                  (. shortcircuit.here(); .)
.

EqlExp<out int type>                  (. int type2;
                                         int op; .)
=  RelExp<out type>
   { EqualOp<out op>
     RelExp<out type2>                (. if (!compatible(type, type2))
                                            SemError("incomparable operands");
                                         type = Entry.boolType; CodeGen.comparison(op); .)
   } .

RelExp<out int type>                  (. int type2;
                                         int op; .)
=  AddExp<out type>
   [ RelOp<out op>
     AddExp<out type2>                (. if (!isArith(type) || !isArith(type2))
                                            SemError("incomparable operands");
                                         type = Entry.boolType; CodeGen.comparison(op); .)
   ] .

AddExp<out int type>                  (. int type2;
                                         int op; .)
=  MultExp<out type>
   { AddOp<out op>
     MultExp<out type2>               (. if (!isArith(type) || !isArith(type2)) {
                                            SemError("arithmetic operands needed");
                                            type = Entry.noType;
                                         }
                                         CodeGen.binaryOp(op); .)
   } .

MultExp<out int type>                 (. int type2;
                                         int op; .)
=  Factor<out type>
   { MulOp<out op>
     Factor<out type2>                (. if (!isArith(type) || !isArith(type2)) {
                                            SemError("arithmetic operands needed");
                                            type = Entry.noType;
                                         }
                                         CodeGen.binaryOp(op); .)
   } .

Factor<out int type>                  (. type = Entry.noType; .)
=    Primary<out type>
   | "+" Factor<out type>             (. if (!isArith(type)) {
                                            SemError("arithmetic operand needed");
                                            type = Entry.noType;
                                         } .)
   | "-" Factor<out type>             (. if (!isArith(type)) {
                                            SemError("arithmetic operand needed");
                                            type = Entry.noType;
                                         }
                                         CodeGen.negateInteger(); .)
   | "!" Factor<out type>             (. if (!isBool(type))
                                            SemError("Boolean operand needed");
                                         type = Entry.boolType; CodeGen.negateBoolean(); .)
.
```

Many submissions had not incorporated the changes needed to handle leading unary + and - in *Factor*.

```
Primary<out int type>                 (. type = Entry.noType;
                                         int size;
                                         DesType des;
                                         ConstRec con; .)
=    Designator<out des>              (. type = des.type;
                                         switch (des.entry.kind) {
                                           case Entry.Var:
                                             CodeGen.dereference();
                                             break;
                                           case Entry.Con:
                                             CodeGen.loadConstant(des.entry.value);
                                             break;
                                           default:
                                             SemError("wrong kind of identifier");
                                             break;
                                         } .)
   | Constant<out con>                (. type = con.type;
                                         CodeGen.loadConstant(con.value); .)
   | "new" BasicType<out type>        (. type++; .)
     "[" Expression<out size>         (. if (!isArith(size))
                                           SemError("array size must be integer");
                                         CodeGen.allocate(); .)
     "]"
   | "(" Expression<out type> ")"
.
```

Notice carefully how the detection of a type incompatibility in some cases is accompanied by forcing the expression or sub-expression to be of the `noType` type, and sometimes of the `boolType` type. This has been done to try to minimize the number of error messages thereafter. You might like to think whether this is a good strategy, or whether it could be improved still further.

We have to refactor the productions defining the various operators in a slightly different way as well:

```
AddOp<out int op>                     (. op = CodeGen.nop; .)
=    "+"                              (. op = CodeGen.add; .)
   | "-"                              (. op = CodeGen.sub; .)
 .

MulOp<out int op>                     (. op = CodeGen.nop; .)
=    "*"                              (. op = CodeGen.mul; .)
   | "/"                              (. op = CodeGen.div; .)
   | "%"                              (. op = CodeGen.rem; .)
.

EqualOp<out int op>                   (. op = CodeGen.nop; .)
=    "=="                             (. op = CodeGen.ceq; .)
   | "!="                             (. op = CodeGen.cne; .)
   | "="                              (. SemError("== intended?"); .)
   | "<>"                             (. SemError("!= intended?"); .)
.

RelOp<out int op>                     (. op = CodeGen.nop; .)
=    "<"                              (. op = CodeGen.clt; .)
   | "<="                             (. op = CodeGen.cle; .)
   | ">"                              (. op = CodeGen.cgt; .)
   | ">="                             (. op = CodeGen.cge; .)
.
```

Task 11 - Better string handling

This is easier than might at first appear. The code generation routine becomes

```
public static void writeString(String str) {
// Generates code to output string stored at known location
  emit(PVM.prns); emit(PVM.indexOfString(str));
}
```

Changes are needed in the PVM emulator. Firstly, it can maintain the string pool by using an *ArrayList* as shown here - note that duplicate strings are not added to the pool a second time.

```
public static ArrayList<String> strPool = new ArrayList<String>();

public static int indexOfString(String str) {
// Add str to string pool if not already there, return its index
  int i;
```

```
    for (i = 0; i < strPool.size(); i++)
      if (str.equals(strPool.get(i))) return i;
    /* if not found */ strPool.add(str); return i;
  }
```

Secondly, the interpretation of the `PVM.prns` opcode has to alter:

```
case PVM.prns:          // string output
  if (tracing) results.write(padding);
  results.write(strPool.get(next()));
  if (tracing) results.writeLine();
  break;
```

The part of the `ListCode` method responsible for displaying strings must decode the string in a different way.

```
case PVM.prns:
  i = (i + 1) % memSize;
  codeFile.write(" \"");
  String str = strPool.get(mem[i]);
  for (j = 0; j < str.length(); j++)
    switch (str.charAt(j)) {
      case '\\' : codeFile.write("\\\\"); break;
      case '\"' : codeFile.write("\\\""); break;
      case '\'' : codeFile.write("\\\'"); break;
      case '\b' : codeFile.write("\\b"); break;
      case '\t' : codeFile.write("\\t"); break;
      case '\n' : codeFile.write("\\n"); break;
      case '\f' : codeFile.write("\\f"); break;
      case '\r' : codeFile.write("\\r"); break;
      default   : codeFile.write(str.charAt(j)); break;
    }
  codeFile.write("\"");
  break;
```

Task 12 - Some ideas are worse than a snake in the grass

Handling the Python-like multiple assignment statement calls for yet another modification to the *Assignment* parser. Essentially the idea is to generate code that will first push a set of destination addresses onto the stack, followed by a matching list of expression values. A special assignment opcode can then arrange to work through the list making the required number of assignments.

```
    Assignment                          (. int expType;
                                           DesType des;
**                                         int count = 0;
**                                         int desCount = 1, expCount = 1;
**                                         ArrayList<Integer> typeList = new ArrayList<Integer>(); .)
    =  Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                             SemError("invalid assignment");
**                                         typeList.add(des.type); .)
       (
**       { "," Designator<out des>      (. if (des.entry.kind != Entry.Var)
**                                             SemError("invalid assignment");
**                                         typeList.add(des.type);
**                                         desCount++; .)
**       }
         AssignOp
**       Expression<out expType>        (. if (count < typeList.size()
**                                              && !compatible(typeList.get(count++), expType))
**                                             SemError("incompatible types in assignment"); .)
**       { "," Expression<out expType>  (. if (count < typeList.size()
**                                              && !compatible(typeList.get(count++), expType))
**                                             SemError("incompatible types in assignment");
**                                         expCount++; .)
**       }                              (. if (expCount != desCount)
**                                             SemError("left and right counts disagree");
**                                         if (desCount == 1)
**                                             CodeGen.assign(des.type);
**                                         else CodeGen.assignN(desCount); .)
         | "++"                         (. if (!isArith(des.type))
                                             SemError("arithmetic type needed");
                                         CodeGen.increment(des.type); .)
         | "--"                         (. if (!isArith(des.type))
                                             SemError("arithmetic type needed");
                                         CodeGen.decrement(des.type); .)
       )
       WEAK ";" .
```

Notice the semantic checks to ensure that the numbers of designators and expressions agree, and that the types of the expressions are compatible with the types of the designators. This last check requires that we build up a list of the designator types - and when we examine this list later we must take care to avoid disaster in retrieving more designator types than have been added to the list! Note that not all the assignments have to be of the same "type", which seems to have been overlooked in some of the submissions that attempted this exercise. The code generator routine is simple:

```
public static void assignN(int n) {
// Generates code to store n values currently on top-of-stack on the n addresses
// below these, finally discarding 2*n elements from the stack.
  emit(PVM.ston); emit(n);
}
```

and interpretation is fairly straightforward:

```
      case PVM.ston:          // store n values at top of stack on addresses below them on stack
        int n = next();
        for (int i = n - 1; i >= 0; i--)
          mem[mem[cpu.sp + n + i]] = mem[cpu.sp + i]; // do the assignments
                                                      // then bump stack pointer to
        cpu.sp = cpu.sp + 2 * n;                      // discard addresses and values
        break;
```

Task 13 - Let's dive in where C programmers fear to go

We add the *in* operator into the hierarchy at the same level as the other relation operators, though of course the syntax is somewhat different:

```
        RelExp<out int type>                 (. int type2;
                                                int op;
                                                int count; .)
        =  AddExp<out type>
           [ RelOp<out op>
                AddExp<out type2>            (. if (!isArith(type) || !isArith(type2))
                                                  SemError("incomparable operands");
                                                type = Entry.boolType; CodeGen.comparison(op); .)
**         | "in" ExpList<out count, type> (. type = Entry.boolType; CodeGen.membership(count); .)
           ] .
```

Parsing the *ExpList* must include the check that all the members of the list are of a type compatible with the expression that is to be tested for membership. This routine returns the number of elements in the list, and generates code to push the values of the expressions onto the stack.

```
**  ExpList<out int count, int type>      (. int type2;
**                                           count = 1; .)
**  =  "("
**        Expression<out type2>           (. if (!compatible(type, type2))
**                                             SemError("incompatible types"); .)
**        { "," Expression<out type2>     (. if (!compatible(type, type2))
**                                             SemError("incompatible types");
**                                           count++; .)
**        }
**     ")" .
```

Note that the compatibility tests are done after each *Expression* has been parsed, and not left until the end. A special opcode is used to test for membership. Code generation is easy:

```
public static void membership(int count) {
// Generates code to check membership of a list of count expressions
  emit(PVM.memb); emit(count);
}
```

while interpretation requires that the value of each of the expressions is popped off the stack and checked to see whether it matches the value being tested for membership. Finally the last value is popped, and a pseudo-boolean value left on top of the stack:

```
case PVM.memb:          // membership test
  boolean isMember = false;
  loop = next();
  int test = mem[cpu.sp + loop];
  for (int l = 0; l < loop; l++) if (pop() == test) isMember = true;
  mem[cpu.sp] = isMember ? 1 : 0;
  break;
```

Task 14 - What are we doing all this for?

This solution includes the possibility of the loop body incorporating one or more *BreakStatements* or *ContinueStatements*, and also handles the problem of the undeclared control variable by entering it into the symbol table.

We make use of the same *ExpList* production. Note particularly where the various labels are defined:

```
ForStatement<StackFrame frame>        (. loopLevel++;
                                         Label oldContinue = loopContinue;
                                         Label oldExit = loopExit;
                                         loopExit = new Label(!known);
                                         loopContinue = new Label(!known);
                                         DesType des;
                                         int count; .)
=  "for" "("
   Designator<out des>                (. if (isRef(des.type))
                                           SemError("invalid control variable"); .)
   "in"
   ExpList<out count, des.type>
   ")"                                (. CodeGen.loadConstant(count);
                                         Label startLoop = new Label(known);
                                         CodeGen.nextControl(count); .)
   Statement<frame>                   (. loopContinue.here();
                                         CodeGen.testFor(startLoop);
                                         loopExit.here();
                                         CodeGen.exitFor(count);
                                         loopExit = oldExit;
                                         loopContinue = oldContinue;
                                         loopLevel--; .) .
```

At run time, just before the *Statement* is executed, the top elements on the stack are set up in a manner that is exemplified by th

```
for (i in (35, 64, -1, 235) write(i);   // loop to be executed 4 times
```

...	4	235	-1	64	35	adr i	....

The 4 on the top of the stack will be used as a counter and decremented on each pass of the loop. The values below that will be assigned, one by one, to the designator whose address is lurking at the bottom of this segment.

The extra code generating methods are as follows:

```
public static void nextControl(int count) {
// Generates ccode to set the for loop control on each pass
  emit(PVM.sfl); emit(count);
}

public static void testFor(Label startLoop) {
// Generates code to decrement the count after each pass through loop
// and test to see whether another iteration is needed
  emit(PVM.tfl); emit(startLoop.address());
}

public static void exitFor(int count) {
// Generates code to discard the expression list used in the for loop
  emit(PVM.efl); emit(count);
}
```

and, finally, the magic that makes this all work effectively is achieved with the new opcodes that are interpreted as follows. Once again, note carefully how and where the various labels are used.

```
case sfl:               // set for loop control variable
  mem[mem[cpu.sp + next() + 1]] = mem[cpu.sp + mem[cpu.sp]];
  break;
case tfl:               // test for loop continuation
  target = next();
  mem[cpu.sp]--;
  if (mem[cpu.sp] > 0) cpu.pc = target;
  break;
case efl:               // end for loop - clean up stack
  cpu.sp += next() + 2;
  break;
```