Computer Science 3 - 2013

Programming Language Translation

Practical for Weeks 25 - 26, beginning 14 October 2013 - 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 - Use of the debugging and other pragmas

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; .)
*     CodeOn       = "$C+" .            (. listCode  = true; .)
*     CodeOff      = "$C-" .            (. listCode  = false; .)
*     WarnOn       = "$W+" .            (. warnings  = true; .)
*     WarnOff      = "$W-" .            (. warnings  = 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 achieved - we make the sub-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, but it is better as an error, I think):

```
    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 introduction of then into an *IfStatement* is quite easily achieved. At this stage it looks like this (but see later tasks).

```
   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 - Things are not always what they seem

Issuing warnings for empty statements or empty blocks at first looks quite easy. At this stage we could try:

```
   Statement<StackFrame frame>          (. boolean empty = false; .)
*  =  SYNC (   Block<frame>
             | ConstDeclarations
             | VarDeclarations<frame>
             | AssignmentStatement
             | IfStatement<frame>
             | WhileStatement<frame>
             | SwitchStatement<frame>
             | HaltStatement
             | ReturnStatement
             | ReadStatement
             | WriteStatement
             | "stackdump" ";"          (. if (debug) CodeGen.dump(); .)
*            | ";"                      (. if (warnings) Warning("empty statement"); .)
   .

*  Block<StackFrame frame>
   =                                    (. Table.openScope();
*                                          bool empty = true; .)
*     "{" { Statement<frame>            (. empty = false; .)
*         }                             (. if (empty && warnings) Warning("empty {} block");
       WEAK "}"                         (. if (debug) Table.printTable(OutFile.StdOut);
                                           Table.closeScope(); .) .
```

Spotting an empty block or the empty statement in the form of a stray semicolon, is partly helpful. Detecting blocks that really have no effect might be handled in several ways. One suggestion is to count the executable statements in a *Block*. This means that the *Statement* parser has to be attributed so as to return this count, and this has a knock-on effect in various other productions as well. Since we might have all sorts of nonsense like

```
   { { int k; } { { int j; } int i; ; ;  { } {{}} } }
```

counting has to proceed carefully. Details are left as a further exercise! Once you have started seeing how stupid some code can be, you can develop a flare for writing bad code suitable for testing compilers without asking your friends in CSC 102 to do it for you!

Task 5 - Something to do - while you wait for inspiration

Adding the basic *DoWhile* loop to Parva is very easy too, since all that is needed is a "backward" branch. Note the use of the `negateBoolean` method, as the PVM does not have a `BNZ` opcode (although it would be easy enough to add one):

```
*  DoWhileStatement<StackFrame frame>   (. Label startLoop = new Label(known); .)
*  =  "do"
*       Statement<frame>
*     WEAK "while"
*     "(" Condition ")" WEAK ";"        (. CodeGen.negateBoolean();
*                                          CodeGen.branchFalse(startLoop); .)
*  .
```

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

The problem, firstly, asked for the addition of an *else* option to the *IfStatement*. Adding an *else* option to the

IfStatement is easy once you see the trick. Note the use of the "no else part" option associated with an action, even in the absence of any terminals or non-terminals. As mentioned earlier, this is a very useful trick to remember.

```
     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>
*       (   "else"                        (. CodeGen.branch(outLabel);
*                                            falseLabel.here(); .)
*           Statement<frame>              (. outLabel.here(); .)
*         | /* no else part */            (. falseLabel.here(); .)
*       ) .
```

Adding the *elsif* clauses calls for a little thought. Here is a nice solution:

```
     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(); .)
     .
```

Many - perhaps most - people in attempting this problem come up with the following sort of thing instead. This can generate `BRN` instructions where none are needed. Devoid of checking, just to save space:

```
     IfStatement<StackFrame frame>        (. Label falseLabel = new Label(!known);
                                             Label outLabel = new Label(!known); .)
     =  "if" "(" Condition ")"            (. CodeGen.branchFalse(falseLabel); .)
          Statement<frame>                (. CodeGen.branch(outLabel);
                                             falseLabel.here(); .)
        { "elsif" "(" Condition ")"       (. falseLabel = new Label(!known);
                                             CodeGen.branchFalse(falseLabel); .)
          Statement<frame>                (. CodeGen.branch(outLabel);
                                             falseLabel.here(); .)
        }
        [ "else"  Statement<frame>  ]     (. outLabel.here(); .) .
```

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    ....
```

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. Trying to find a context-free grammar with this restriction is not worth the effort.

One approach that incorporates context-sensitive checking in conjunction with code generation is based on passing labels as arguments to various subparsers. We change the parser for *Statement* and for *Block* as follows:

```
*  Statement<StackFrame frame, Label breakLabel>
*  =  SYNC (   Block<frame, breakLabel>
             | ConstDeclarations
             | VarDeclarations<frame>
             | AssignmentStatement
*            | IfStatement<frame, breakLabel>
             | WhileStatement<frame>
             | DoWhileStatement<frame>
             | SwitchStatement<frame>
*            | BreakStatement<breakLabel>
             | HaltStatement
             | ReturnStatement
             | ReadStatement
             | ReadLineStatement
             | WriteStatement
             | WriteLineStatement
             | "stackdump" ";"           (. if (debug) CodeGen.dump(); .)
             | ";"                       (. if (warnings) Warning("empty statement"); .)
           ) .

*  Block<StackFrame frame, Label breakLabel>
   =                                     (. Table.openScope();
                                            bool empty = true; .)
*     "{" { Statement<frame, breakLabel> (. empty = false; .)
          }                              (. if (empty && warnings) Warning("empty block"); .)
      WEAK "}"                           (. if (debug) Table.printTable(OutFile.StdOut);
                                            Table.closeScope(); .) .
```

and the parsers for the statements that are concerned with looping, breaking, and making decisions become

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

   WhileStatement<StackFrame frame>      (. Label loopExit  = new Label(!known);
                                            Label loopStart = new Label(known); .)
   =  "while" "(" Condition ")"          (. CodeGen.branchFalse(loopExit); .)
*     Statement<frame, loopExit>         (. CodeGen.branch(loopStart);
                                            loopExit.here(); .) .

   BreakStatement<Label breakLabel>
*  =  "break"                            (. if (breakLabel == null)
*                                              SemError("break is not allowed here");
*                                           else CodeGen.branch(breakLabel); .)
      WEAK ";" .

   DoWhileStatement<StackFrame frame>    (. Label loopExit = new Label(!known);
                                            Label loopStart = new Label(known); .)
   =  "do"
*       Statement<frame, loopExit>
      WEAK "while"
      "(" Condition ")" WEAK ";"         (. CodeGen.negateBoolean();
                                            CodeGen.branchFalse(loopStart);
*                                           loopExit.here(); .) .
```

There is at least one other way of solving the problem, which involves building one's own stack of labels and maintaining it outside of the methods. But the method given here effectively does it automagically, building up the label stack in the stack frames that are constructed as the methods are called (remember the discussion in class about how methods are called and build stack frames …?)

Task 8 - 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 at the initial level 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 and for the designation of variables rather than constants that several students omitted) by assuming the existence of a few new machine opcodes, as suggested in the textbook.

```
   Assignment                          (. int expType;
                                          DesType des;
*                                         boolean inc = true; .)
   =    Designator<out des>            (. if (des.entry.kind != Kinds.Var)
                                            SemError("invalid assignment"); .)
        (   AssignOp
            Expression<out expType>    (. if (!assignable(des.type, expType))
                                            SemError("incompatible types in assignment");
                                          CodeGen.assign(des.type); .)
*         | ( "++" | "--"              (. inc = false; .)
*           )                          (. if (!isArith(des.type))
*                                           SemError("arithmetic type needed");
*                                         CodeGen.incOrDec(inc, des.type); .)
*       )
*    |  ( "++" | "--"                  (. inc = false; .)
*       ) Designator<out des>          (. if (des.entry.kind != Kinds.Var)
*                                           SemError("variable designator required");
*                                         if (!isArith(des.type))
*                                           SemError("arithmetic type needed");
*                                         CodeGen.incOrDec(inc, des.type); .) .
      WEAK ";" .
```

The extra code generation routine is straightforward, but note that we need to cater for characters specially

```
   public static void incOrDec(boolean inc, int type) {
   // Generates code to increment the value found at the address currently
   // stored at the top of the stack.
   // If necessary, apply character range check
*    if (type == Types.charType) emit(inc ? PVM.incc : PVM.decc);
*    else emit(inc ? PVM.inc : 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;
   case PVM.incc:          // char ++
     adr = pop();
     if (inBounds(adr))
       if (mem[adr] < maxChar) mem[adr]++;
       else ps = badVal;
     break;
   case PVM.decc:          // char --
     adr = pop();
     if (inBounds(adr))
       if (mem[adr] > 0) mem[adr]--;
       else ps = badVal;
     break;
```

Task 9 - A short circuit does not always signify a quick trip around the Prospect Field track

The exercise suggested that the user might be allowed to use a `$S` pragma or `-s` command line parameter to choose between code generation using short-circuit semantics or code generation using a Boolean operator approach (see the textbook, pages 12 and 167). All the opcodes you need are already in the source kit.

This is easily implemented as follows, where we have also shown how the feature might be controlled by a Boolean flag set by the pragma:

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

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

Task 10 - It should only take a MIN or two to derive MAX benefit from these tasks

The problem called for extensions to the grammar, code generator and the PVM to allow you to incorporate calls to `max()` or `min()` functions. The most obvious use of these might be limited to two arguments, as in

```
min(a, b) - max(c, d)
```

but in general one should be able to deal with any number of arguments:

```
min(a, b) - max(c, d) + min(e) + max(w, x, y, z) + max(min(e, f + max(p, q)))
```

Once again, this is all easily achieved by additions to the options in the *Primary* production. One way of doing this is to generate code for each argument (expression) and then to follow this by a call to a new code generating function. Note the auto-promotion to integer type if any of the arguments are of the integer type.

```
| (    "max"                        (. max = true; .)
     | "min"                        (. max = false; .)
  )
  "("
     Expression<out type>           (. if (!isArith(type))
                                         SemError("arithmetic argument expected"); .)
     { "," Expression<out type2>    (. if (!isArith(type2))
                                         SemError("arithmetic argument expected");
                                       else if (type2 != Types.charType) type = type2;
                                       CodeGen.maxMin(max); .)
     }
  ")"
```

The code generator and the PVM are easily extended

```
public static void maxMin(boolean max) {
// Generates code to leave max/min(tos, sos) on top of stack
  emit(max ? PVM.max : PVM.min);
}

case PVM.max:            // max(tos, sos)
  tos = pop();
  sos = pop();
  push(tos > sos? tos : sos);
  break;
```

Note that this still will work for the pathological case where the `max()` or `min()` function has only one argument! There are other simple changes needed to the PVM - the new opcodes must be added to the opcode list, and must have appropriate mnemonics defined. The changes needed here - and in the plethora of similar changes needed in some of these exercises - are straightforward and can all be seen in the source kit.

There is another approach - one could generate code to push the values of all of the arguments onto the run-time stack, counting them at the same time, and then generate a two-word opcode to be used by the emulator. This would suggest changes to the *Primary* production as follows:

```
| (   "MAX"                       (. max = true; .)
    | "MIN"                       (. max = false; .)
  )
  "("
     Expression<out type>         (. int count = 1;
                                     if (!isArith(type))
                                       SemError("arithmetic argument expected"); .)
     { "," Expression<out type2>  (. count++;
                                     if (!isArith(type))
                                       SemError("arithmetic argument expected"); .)
                                     else if (type2 != Types.charType) type = type2;
     }
  ")"                             (. CodeGen.maxMin(max, count); .)
```

along with a code generator method as follows:

```
public static void maxMin(boolean max, int count) {
// Generates code to leave max(a,b,c ... ) of count values on top of stack
  emit(max ? PVM.max2 : PVM.min2); emit(count);
}
```

and emulation on the lines of the following (with a similar idea for finding the minimum):

```
case PVM.max2:          // max(a,b,c....)
  loop = next();
  while (loop > 1) {
    tos = pop();
    sos = pop();
    push(tos > sos? tos : sos);
    loop--;
  }
  break;
```

Task 11 - In case you have nothing better to do (Switch to Parva - Success Guaranteed)

The problem called for the implementation of a *SwitchStatement* described by the productions:

```
SwitchStatement
=  "switch" "(" Expression ")" "{"
      { CaseLabelList Statement { Statement } }
      [ "default" ":" { Statement } ]
   "}" .
CaseLabelList = CaseLabel { CaseLabel } .
CaseLabel     = "case" [ "+" | "-" ] Constant ":" .
```

as exemplified by

```
switch (i + j) {
  case 2   : if (i === j) break; write("i = " , i); read(i, j);
  case 4   : write("four"); i = 12;
  case 6   : write("six");
  case -9  :
  case 9   :
  case -10 :
  case 10  : write("plus or minus nine or ten"); i = 12;
  default  : write("not 2, 4, 6, 9 or 10");
}
```

by generating code matching an equivalent set of *IfStatement*s, effectively on the lines of

```
      temp = i + j;
      if    (temp == 2) { if (i === j) goto out; write("i = " , i); read(i, j); goto out; }
      elsif (temp == 4) { write("four"); i = 12; goto out; }
      elsif (temp == 6) { write("six"); goto out; }
      elsif (temp in (-9, 9, -10, 10)) { write("plus or minus nine or ten"); i = 12; goto out; }
      else write("not 2, 4, 6, 9 or 10");
  out: ...
```

The `temp` value needed can be stored on the stack - if we execute a DUP opcode before each successive comparison or test for list membership is effected, we can ensure that the value of the selector is preserved, in readiness for the next comparison.

Although this idea does not lead to a highly efficient implementation of the *SwitchStatement*, it is relatively easy to implement - the complexity arising from the need, as usual, to impose semantic checks that all labels are unique, that the type of the selector is compatible with the type of each label, and from a desire to keep the number of branching operations as low as possible. The code follows:

```
SwitchStatement<StackFrame frame>     (. int expType, expCount, labelCount;
                                         boolean branchNeeded = false;
                                         Label nextSwitch = new Label(!known);
                                         Label switchExit = new Label(!known);
                                         ArrayList<Integer> labelList = new ArrayList<Integer>();  .)
=  "switch" "("
     Expression<out expType>          (. if (isRef(expType) || expType == Types.noType)
                                           SemError("invalid selector type"); .)
   ")"
   "{"
      {                               (. if (branchNeeded) CodeGen.branch(switchExit);
                                         branchNeeded = true;
                                         nextSwitch.here();
                                         nextSwitch = new Label(!known);
                                         CodeGen.duplicate(); .)
        CaseLabelList<out labelCount, expType, labelList>
                                      (. CodeGen.membership(labelCount, expType);
                                         CodeGen.branchFalse(nextSwitch); .)
        Statement<frame, switchExit>
        { Statement<frame, switchExit> }
      }
      (   "default" ":"               (. if (branchNeeded) CodeGen.branch(switchExit);
                                         nextSwitch.here(); .)
         { Statement<frame, switchExit> }
        |                             (. nextSwitch.here(); .)
      )
   "}"                                (. switchExit.here();
                                         CodeGen.pop(1); .) .


CaseLabelList<. out int labelCount, int expType, ArrayList<Integer> labelList .>
=  CaseLabel<expType, labelList>      (. labelCount = 1; .)
   { CaseLabel<expType, labelList>    (. labelCount++; .)
   } .

CaseLabel<. int expType, ArrayList<Integer> labelList .>
                                      (. ConstRec con;
                                         int factor = 1;
                                         boolean signed = false; .)
=  "case"
   [ ( "+" | "-"                      (. factor = -1; .)
     )                                (. signed = true; .)
   ] Constant<out con> ":"            (. if (!compatible(con.type, expType)
                                              || signed && con.type != Types.intType)
                                           SemError("invalid label type");
                                         int lab = factor * con.value;
                                         if (labelList.contains(lab))
                                           SemError("duplicated case label");
                                         else labelList.add(lab);
                                         CodeGen.loadConstant(lab); .) .
```

Notes

- Note the use of the `<. ... .>` bracketing around the parameter lists for the *CaseLabelList* and *CaseLabel* productions. There are needed because of the syntax required for dealing with generic classes in Java and C#.

- Each *SwitchStatement* implements a simple list for recording the values of its labels - which must be unique within a single *SwitchStatement*. We cannot use a global or static field in the parser, so this structure has to be passed down the chain of calls to other routines.

- The use of the `branchNeeded` variable is to ensure that the minimum number of branch operations is introduced. It is possible to find other actions that do not use this variable, but (as in the case of the non-optimal *IfStatement* discussed earlier) these may have the effect of creating unnecessary branches from one operation straight to the next.

- The system correctly handles a *SwitchStatement* with case labels but no default option, with no case labels and only a default option, or even with none of them at all!

- There is a school of thought that suggests that, in the absence of an explicit default option, failure to match a case label should simply "do nothing" is dangerous practice, and that one should always be required to supply one - even if the associated statement list is missing. However, it would be very easy to modify the grammar above to achieve this.

- Note that the statement sequences within a *SwitchStatement* might incorporate one or more explicit *BreakStatement*s. These, of course, are distinct from any *BreakStatements* that might be used to exit loops within the statement sequences, but the mechanism suggested here handles this correctly. It also effectively forbids stray *ContinueStatement*s from appearing. You might like to consider whether the *ContinueStatement* could be used as a means of providing the "fall through" semantics of some other versions of the *SwitchStatement*.

- The form of code generated by this system may be understood by reference to the following example

```
switch (selector) {
  case 20: case 30: case 40: statement 1;
  case 50: statement 2;
  default: statement 3;
}
```

which gives rise to code like

```
        selector
        DUP
        LDC     20
        LDC     30
        LDC     40
        MEMB    3
        BZE     next
        statement 1
        BRN     exit
next    DUP
        LDC     50
        CEQ
        BZE     default
        statement 2
        BRN     exit
default statement 3
exit    DSP     -1
        ...
```

This calls for other simple opcodes for the PVM. DUP can be generated by calling:

```
public static void duplicate() {
// Generates code to push another copy of top of stack
  emit(PVM.dup);
}
```

and its interpretation is as follows:

```
case PVM.dup:           // duplicate top of stack
  cpu.sp--;
  if (inBounds(cpu.sp)) mem[cpu.sp] = mem[cpu.sp + 1];
  break;
```

`PVM.memb` can be generated (when there are two or more labels) by calling

```
public static void membership(int count, int type) {
// Generates code to check membership of a list of count expressions
  if (count == 1) comparison(CodeGen.ceq, type);
  else { emit(PVM.memb); emit(count); }
}
```

with a two-word opcode interpreted as follows:

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

Task 12 - Generating tighter PVM code

The changes to the code generating routines to produce the special one-word opcodes like `LDA_0` and `LDC_3` and the others like them are very simple, on the lines of the following:

```
public static void loadConstant(int number) {
// Generates code to push number onto evaluation stack
  switch (number) {
    case -1: emit(PVM.ldc_m1); break;
    case 0:  emit(PVM.ldc_0); break;
    case 1:  emit(PVM.ldc_1); break;
    case 2:  emit(PVM.ldc_2); break;
    case 3:  emit(PVM.ldc_3); break;
    case 4:  emit(PVM.ldc_4); break;
    case 5:  emit(PVM.ldc_5); break;
    default: emit(PVM.ldc); emit(number); break;
  }
}

public static void loadAddress(Entry var) {
// Generates code to push address of variable var onto evaluation stack
  switch (var.offset) {
    case 0:  emit(PVM.lda_0); break;
    case 1:  emit(PVM.lda_1); break;
    case 2:  emit(PVM.lda_2); break;
    case 3:  emit(PVM.lda_3); break;
    case 4:  emit(PVM.lda_4); break;
    case 5:  emit(PVM.lda_5); break;
    default: emit(PVM.lda); emit(var.offset); break;
  }
}
```

Of course, with the Parva grammar as it was defined for this practical one would never be in a position to generate the `ldc_m1` opcode, since the grammar made no provision for negative constants. It would not have been hard to extend it to do so, and you might like to puzzle out how and where this could be done.

As stated in the prac sheet, generating code to make use of `LDL` and `STL` is something that must be done with great care. Various of the productions -*Assignment, OneVar, Designator* and *Primary* need alteration. The trick is to modify the *Designator* production so that it does not generate the `LDA` opcode immediately. But we need to distinguish between designators that correspond to "simple" variables that are to be manipulated with the `LDL` and `STL` opcodes, and array elements which will still require use of `LDV` and `STO` opcodes. So the *DesType* class is extended:

```
    class DesType {
    // Objects of this type are associated with l-value and r-value designators
      public Entry entry;          // the identifier properties
      public int type;             // designator type (not always the entry type)
*     public boolean isSimple;     // true unless it is an indexed designator

      public DesType(Entry entry) {
        this.entry = entry;
        this.type = entry.type;
*       this.isSimple = true;
      }
    } // end DesType
```

The *Designator* production is now attributed as follows - note in particular where the code generation occurs:

```
   Designator<out DesType des>          (. string name;
                                           int indexType; .)
   =  Ident<out name>                   (. Entry entry = Table.find(name);
                                           if (!entry.declared)
                                             SemError("undeclared identifier");
                                           des = new DesType(entry); .)
      [  "["                            (. if (isRef(des.type)) des.type--;
                                           else SemError("unexpected subscript");
                                           if (entry.kind != Kinds.Var)
                                             SemError("unexpected subscript");
*                                          des.isSimple = false;
*                                          CodeGen.loadValue(entry); .)
            Expression<out indexType>   (. if (!isArith(indexType)) SemError("invalid subscript type");
*                                          CodeGen.index(); .)
         "]"
      ] .
```

Within the *Primary* production, when a *Designator* is parsed one must either complete the array access by generating the LDV opcode, or generate the LDL opcode.

```
   Primary<out int type>                (. type = Types.noType;
                                           int size;
                                           DesType des;
                                           ConstRec con; .)
   =    Designator<out des>             (. type = des.type;
                                           switch (des.entry.kind) {
                                             case Kinds.Var:
*                                              if (des.isSimple) CodeGen.loadValue(des.entry);
*                                              else CodeGen.dereference();
                                               break;
                                             case Kinds.Con:
                                               CodeGen.loadConstant(des.entry.value);
                                               break;
                                             default:
                                               SemError("wrong kind of identifier");
                                               break;
                                           } .)
      | Constant<out con> ... // as before  .
```

When variables are declared we can always make use of the STL code if they are initialized:

```
   OneVar<StackFrame frame, int type>   (. int expType; .)
   =                                    (. Entry var = new Entry(); .)
      Ident<out var.name>               (. var.kind = Kinds.Var;
                                           var.type = type;
                                           var.offset = frame.size;
                                           frame.size++; .)
      [ AssignOp Expression<out expType> (. if (!asssignable(var.type, expType))
                                             SemError("incompatible types in assignment");
*                                          CodeGen.storeValue(var); .)
      ]                                 (. Table.insert(var); .) .
```

The production for *ReadElement* will have to generate the LDA opcode if the element to be read is a simple variable:

```
   ReadElement                          (. string str;
                                           DesType des; .)
   =   StringConst<out str>             (. CodeGen.writeString(str); .)
     | Designator<out des>              (. if (des.entry.kind != Kinds.Var)
                                             SemError("wrong kind of identifier");
*                                          if (des.isSimple) CodeGen.loadAddress(des.entry);
                                           switch (des.type) {
                                           ...  // as before
```

Similarly, the production for *Assignment* may have to generate the LDA opcode if the ++ or -- operation is applied to simple variables, and to choose between generating the STL or STO opcodes for regular assignment statements.

Task 13 - Peek and Poke

In the extension to Parva to allow the compiler to download code into the PAM computer it was suggested that you add the ability to read directly from any addressible location in memory (peek) or to write directly to any addressible location in memory (poke).

Adding these extensions to Parva should have been no real challenge for students who had got this far. The production for `PokeStatement` is as follows:

```
PokeStatement                        (. int type, address;
                                        ConstRec con; .)
=  "poke" "(" Expression<out type>   (. if (!isArith(type))
                                          SemError("integer address needed"); .)
    WEAK "," Expression<out type>    (. if (!isArith(type))
                                          SemError("integer expression needed");
                                        CodeGen.assign(type); .)
   ")" WEAK ";" .
```

and the extra possibility within *Factor* is as follows:

```
   | "peek"
     "(" Expression<out type>        (. if (!isArith(type))
                                          SemError("Arithmetic argument needed");
                                        CodeGen.dereference();
                                        type = Types.intType; .)
```

Unchecked, these are fairly "dangerous" facilities (why?). It is left as an further exercise to improve security.