Computer Science 3 - 2011

Programming Language Translation

Practical for Weeks 25 - 26, beginning 17 October 2011 - 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<out empty, frame>
             | ConstDeclarations
             | VarDeclarations<frame>
             | AssignmentStatement
             | IfStatement<frame>
             | WhileStatement<frame>
             | HaltStatement
             | ReturnStatement
             | ReadStatement
             | WriteStatement
*            | ";"                       (. empty = true; .)
             | "stackdump" ";"           (. if (debug) CodeGen.dump(); .)
*         )                              (. if (empty && warnings)
*                                             Warning("empty statement"); .)
   .

*  Block<out boolean empty, StackFrame frame>
   =                                     (. Table.openScope();
*                                           empty = true; .)
*     "{" { Statement<frame>             (. empty = false; .)
*         } WEAK "}"                     (. if (empty && warnings)
*                                             Warning("empty {} block");
                                          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. The suggestion below counts 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. 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!

```
   Statement<out int execCount, StackFrame frame>
                                         (. execCount = 0; .)
   =  SYNC (   Block<out execCount, frame>
             | ConstDeclarations
             | VarDeclarations<frame>
*            | ";"                       (. if (warnings) Warning("empty statement"); .)
*            |                           (. execCount = 1; .)
               (   Assignment
                 | IfStatement<frame>
                 | WhileStatement<frame>
                 | DoWhileStatement<frame>
                 | BreakStatement
                 | HaltStatement
                 | ReturnStatement
                 | ReadStatement
                 | ReadLineStatement
                 | WriteStatement
                 | WriteLineStatement
                 | "stackdump" ";"       (. if (debug) CodeGen.dump(); .)
               )
           ) .
```

```
   Block<out int execCount, StackFrame frame>
*  =                                    (. int count = 0;
*                                          execCount = 0;
                                           Table.openScope(); .)
      "{"
*     { Statement<out count, frame>     (. execCount += count; .)
      }
*     WEAK "}"                          (. if (execCount == 0 && warnings)
                                             Warning("no executable statements in block");
                                           if (debug) Table.printTable(OutFile.StdOut);
                                           Table.closeScope(); .) .
```

A similar modification is needed in the *Parva* production, which you can study in the full source code.

Task 5 - Two of the three Rs - Reading and Writing

The extensions to the *ReadStatement* and *WriteStatement* are very simple. It is useful to allow both the `readLine()` and `writeLine()` statements to have empty argument lists, something a little meaningless for the former *Read* and *Write* statements. The extensions needed to the code generator and the PVM should be obvious, but it was clear from some submissions that some of you are unfamiliar with the concept of `readLine();` as a device for ignoring the rest of a line in the *data* file (not, for heaven's sake, in the grammar!)

```
*  ReadLineStatement /* optional arguments! */
*  = "readLine" "(" [ ReadElement { WEAK "," ReadElement } ] ")" WEAK ";"
*                                    (. CodeGen.readLine(); .) .

*  WriteLineStatement /* optional arguments! */
*  = "writeLine" "(" [ WriteElement { WEAK "," WriteElement } ] ")" WEAK ";"
*                                    (. CodeGen.writeLine(); .) .
```

Task 6 - 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>   (. int count;
*                                          Label startLoop = new Label(known); .)
*  =  "do"
*        Statement<out count, frame>    (. if (count == 0 && warnings)
*                                            Warning("empty statement part"); .)
*     WEAK "while"
*     "(" Condition ")" WEAK ";"        (. CodeGen.negateBoolean();
*                                          CodeGen.branchFalse(startLoop); .)
*  .
```

Task 7 - 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>        (. int count;
                                           Label falseLabel = new Label(!known);
                                           Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"            (. CodeGen.branchFalse(falseLabel); .)
      [ "then"                          (. SemError("then is not used in Parva"); .)
*     ] Statement<out count, frame>     (. if (count == 0 && warnings)
*                                            Warning("empty statement part"); .)
*     (   "else"                        (. CodeGen.branch(outLabel);
*                                          falseLabel.here(); .)
*         Statement<frame, count>       (. if (count == 0 && warnings)
*                                            Warning("empty statement part");
*                                          outLabel.here(); .)
*       | /* no else part */            (. falseLabel.here(); .)
*     ) .
```

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

```
   IfStatement<StackFrame frame>          (. int count;
                                             Label falseLabel = new Label(!known);
                                             Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"              (. CodeGen.branchFalse(falseLabel); .)
      [ "then"                            (. SemError("then is not used in Parva"); .)
      ] Statement<out count, frame>       (. if (count == 0 && warnings)
*                                               Warning("empty statement part"); .)
*     {                                   (. CodeGen.branch(outLabel);
*                                             falseLabel.here();
*                                             falseLabel = new Label(!known); .)
*       "elsif" "(" Condition ")"         (. CodeGen.branchFalse(falseLabel); .)
*       [ "then"                          (. SemError("then is not used in Parva"); .)
*       ] Statement<out count, frame>     (. if (count == 0 && warnings)
*                                               Warning("empty statement part"); .)
*     }
*     (   "else"                          (. CodeGen.branch(outLabel);
*                                             falseLabel.here(); .)
*         Statement<out count, frame>     (. if (count == 0 && warnings)
*                                               Warning("empty statement part"); .)
*       | /* 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 8 - 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<out int execCount, StackFrame frame, Label breakLabel>
                                          (. execCount = 0; .)
*  =  SYNC (   Block<out execCount, frame, breakLabel>
             | ConstDeclarations
             | VarDeclarations<frame>
             | ";"                        (. if (warnings) Warning("empty statement"); .)
             |                            (. execCount = 1; .)
               (   AssignmentStatement
*                | IfStatement<frame, breakLabel>
```

```
                | WhileStatement<frame>
                | DoWhileStatement<frame>
                | ForStatement<frame>
*               | BreakStatement<breakLabel>
                | HaltStatement
                | ReturnStatement
                | ReadLineStatement
                | ReadStatement
                | WriteStatement
                | WriteLineStatement
                | "stackdump" ";"       (. if (debug) CodeGen.dump(); .)
            )
          ) .

*  Block<out int execCount, StackFrame frame, Label breakLabel>
   =                                    (. int count = 0;
                                           execCount = 0;
                                           Table.openScope(); .)
      "{"
*        { Statement<out count, frame, breakLabel>
                                        (. execCount += count; .)
         }
      WEAK "}"                          (. if (execCount == 0 && warnings)
                                             Warning("no executable statements in block");
                                           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>
                                        (. int count;
                                           Label falseLabel = new Label(!known);
                                           Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"            (. CodeGen.branchFalse(falseLabel); .)
      [ "then"                          (. SemError("then is not used in Parva"); .)
*     ] Statement<out count, frame, breakLabel>
                                        (. if (count == 0 && warnings)
                                             Warning("empty statement part");
                                           CodeGen.branch(outLabel);
                                           falseLabel.here(); .)
      { "elsif" "(" Condition ")"       (. falseLabel = new Label(!known);
                                           CodeGen.branchFalse(falseLabel); .)
        [ "then"                        (. SemError("then is not used in Parva"); .)
*       ] Statement<out count, frame, breakLabel>
                                        (. if (count == 0 && warnings)
                                             Warning("empty statement part");
                                           CodeGen.branch(outLabel);
                                           falseLabel.here(); .)
      }
      [ "else"
*       Statement<out count, frame, breakLabel>
                                        (. if (count == 0 && warnings)
                                             Warning("empty statement part"); .)
      ]                                 (. outLabel.here(); .)
   .

   WhileStatement<StackFrame frame>     (. int count;
                                           Label loopExit  = new Label(!known);
                                           Label loopStart = new Label(known); .)
   =  "while" "(" Condition ")"         (. CodeGen.branchFalse(loopExit); .)
*     Statement<out count, frame, loopExit>
                                        (. if (count == 0 && warnings)
                                             Warning("empty statement part");
                                           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>    (. int count;
*                                           Label loopExit = new Label(!known);
                                            Label loopStart = new Label(known); .)
   =  "do"
*       Statement<out count, frame, loopExit>
                                         (. if (count == 0 && warnings)
                                              Warning("empty statement part"); .)
      WEAK "while"
      "(" Condition ")" WEAK ";"         (. CodeGen.negateBoolean();
                                            CodeGen.branchFalse(loopStart);
*                                           loopExit.here(); .) .
```

There is at least one other way of solving the problem, which involves using local variable in the parsing methods to "stacK' up the old label, assign a new one, and then restore the old one afterwards. Only one person trid this 9and got it correct). The fact that so many other people used the method given here suggests to me, unfortunately, that this solution was passed around more than it should have been.

Task 9 - Your professor is quite a character

To allow for a character type involves one in a lot of straightforward alterations, as well as some more elusive ones, and few if any students gor it completely correct. Firstly, we extend the definition of a symbol table entry:

```
   class Entry {
     public static final int
       Con = 0,                  // identifier kinds
       Var = 1,
       Fun = 2,

       noType   =  0,            // identifier (and expression) types.  The numbering is
       nullType =  2,            // significant as array types are denoted by these
       intType  =  4,            // numbers + 1
       boolType =  6,
*      charType =  8,
*      voidType = 10;
     ...
   } // end Entry
```

The *Table* class requires a similar small change to introduce the new type name needed if the symbol table is to be displayed:

```
 *  typeNames.Add("char");
 *  typeNames.Add("char[]");
```

A minor change to the *Constant* production is needed to allow character literals to be regarded as of the new `charType`:

```
   Constant<out ConstRec con>           (. con = new ConstRec(); .)
   =   IntConst<out con.value>          (. con.type = Entry.intType; .)
*    | CharConst<out con.value>         (. con.type = Entry.charType; .)
     | "true"                           (. con.type = Entry.boolType; con.value = 1; .)
     | "false"                          (. con.type = Entry.boolType; con.value = 0; .)
     | "null"                           (. con.type = Entry.nullType; con.value = 0; .) .
```

Reading and writing single characters is easy:

```
   ReadElement                          (. String str;
                                           DesType des; .)
*  =   StringConst<out str>             (. CodeGen.writeStr(str); .)
     | Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                             SemError("wrong kind of identifier");
*                                          if (!des.canChange)
*                                            SemError("may not alter this variable");
                                           switch (des.type) {
                                             case Entry.intType:
                                             case Entry.boolType:
*                                            case Entry.charType:
                                               CodeGen.read(des.type); break;
                                             default:
                                               SemError("cannot read this type"); break;
                                           } .) .
```

```
   WriteElement                          (. int expType;
                                            String str; .)
*  =   StringConst<out str>              (. CodeGen.writeStr(str); .)
     | Expression<out expType>           (. switch (expType) {
                                              case Entry.intType:
                                              case Entry.boolType:
*                                             case Entry.charType:
                                                CodeGen.write(expType); break;
                                              default:
                                                SemError("cannot write this type"); break;
                                            } .) .
```

The associated code generating methods require matching additions:

```
   public static void read(int type) {
   // Generates code to read a value of specified type
   // and store it at the address found on top of stack
     switch (type) {
       case Entry.intType:  emit(PVM.inpi); break;
       case Entry.boolType: emit(PVM.inpb); break;
*      case Entry.charType: emit(PVM.inpc); break;
     }
   }

   public static void write(int type) {
   // Generates code to output value of specified type from top of stack
     switch (type) {
       case Entry.intType:  emit(PVM.prni); break;
       case Entry.boolType: emit(PVM.prnb); break;
*      case Entry.charType: emit(PVM.prnc); break;
     }
   }
```

The major part of this exercise was concerned with the changes needed to apply various constraints on operands of the `char` type. Essentially it ranks as an arithmetic type, in that expressions of the form

character + character
character > character
character + integer
character > integer

are all allowable. This can be handled by modifying the helper methods in the parser as follows:

```
   static boolean isArith(int type) {
*    return type == Entry.intType || type == Entry.charType || type == Entry.noType;
   }

*  static boolean compatible(int typeOne, int typeTwo) {
*  // Returns true if typeOne is compatible (and comparable for equality) with typeTwo
     return    typeOne == typeTwo
*            || isArith(typeOne) && isArith(typeTwo)
             || typeOne == Entry.noType || typeTwo == Entry.noType
             || isRef(typeOne) && typeTwo == Entry.nullType
             || isRef(typeTwo) && typeOne == Entry.nullType;
   }
```

However, assignment compatibility is more restrictive. Assignments of the form

integer = integer expression
integer = character expression
character = character expression

are allowed, but

character = integer expression

is not allowed. This may be checked within the *Assignment* production with the aid of a further helper method `assignable`:

```
*  static boolean assignable(int typeOne, int typeTwo) {
*  // Returns true if typeOne may be assigned a value of typeTwo
*    return    typeOne == typeTwo
*           || typeOne == Entry.intType && typeTwo == Entry.charType
*           || typeOne == Entry.noType || typeTwo == Entry.noType
*           || isRef(typeOne) && typeTwo == Entry.nullType;
*  }
```

The `assignable()` function call now takes the place of the `compatible()` function call in the many places in *OneVar* and *Assignment* where, previously, calls to `compatible()` appeared.

We turn finally to consideration of the changes needed to the various sub-parsers for expressions.

A casting mechanism is introduced to handle the situations where it is necessary explicitly to convert integer values to characters, so that

character = (char) integer

is allowed, and for completeness, so are

integer = (int) character
integer = (char) character
character = (char) character

Casting operations are accompanied by a type check and a type conversion; the `(char)` cast also introduces the generation of run-time code for checking that the integer value to be converted lies within range.

This is all handled within the *Primary* production, which has to be factored to deal with the potential LL(1) trap in distinguishing between components of the form `"(" "char" ")"` and `"(" Expression ")"`:

```
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(); .)
     "]"
*  | "("
*    (   "char" ")"
*        Factor<out type>            (. if (!isArith(type))
*                                         SemError("invalid cast");
*                                       else type = Entry.charType;
*                                       CodeGen.castToChar(); .)
*      | "int" ")"
*        Factor<out type>            (. if (!isArith(type))
*                                         SemError("invalid cast");
*                                       else type = Entry.intType; .)
*      | Expression<out type> ")"
*    ) .
```

Strictly speaking the above grammar departs slightly from the Java version, where the casting operator is regarded as weaker than the parentheses around an *Expression*, but in practice it makes little difference.

Various of the other productions need modification. The presence of an arithmetic operator correctly placed

between character or integer operands must result in the sub-expression so formed being of integer type (and never of character type). So, for example:

```
   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;
                                           }
*                                          else type = Entry.intType;
                                           CodeGen.binaryOp(op); .)
      } .
```

Similarly a prefix + or - operator applied to an integer or character *Factor* creates a new factor of integer type (see full solution for details).

The extra code generation method we need is as follows:

```
public static void castToChar() {
// Generates code to check that TOS is within the range of the character type
  emit(PVM.i2c);
}
```

and within the `switch` statement of the `emulator` method we need:

```
case PVM.i2c:           // check convert character to integer
  if (mem[cpu.sp] < 0 || mem[cpu.sp] > maxChar) ps = badVal;
  break;
```

The interpreter has another opcode for checked storage of characters, but if the `i2c` opcodes are inserted correctly it appears that we do not really need `stoc`:

```
case PVM.stoc:          // character checked store
  tos = pop(); adr = pop();
  if (inBounds(adr))
    if (tos >= 0 && tos <= maxChar) mem[adr] = tos;
    else ps = badVal;
  break;
```

Task 10 - 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 != Entry.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 != Entry.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 == Entry.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 11 - 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 is any of the arguments are of type integer.

```
| (   "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 != Entry.charType) type = type2;
                                     CodeGen.maxMin(max); .)
     }
  ")"
```

The code generator is 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);
}
```

and the PVM is extended to have another option in the emulator for the `max` operation and a similar one (not shown) for the `min` operation.

```
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 != Entry.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 12 - What are we doing this for?

Many languages provide for a *ForStatement* in one or other form. Although most people are familiar with these, their semantics and implementation can actually be quite tricky.

This exercise suggested adding a simple Pascal-style *ForStatement* to Parva, to allow statements whose concrete syntax is defined by

```
ForStatement = "for" Ident "=" Expression ("to" | "downto") Expression Statement .
```

The problem as posed already suggested part of a solution. Pascal was a much "safer" language than C, and the semantics of the Pascal-style *ForStatement* are better described as follows. The statements

```
for Control = Expression1 to Expression2  Statement
for Control = Expression1 downto Expression2  Statement
```

should be regarded as more closely equivalent to

```
        Temp1 := Expression1                        Temp1 := Expression1
        Temp2 := Expression2                        Temp2 := Expression2
        IF Temp1 > Temp2 THEN GOTO EXIT             IF Temp1 < Temp2 THEN GOTO EXIT
        Control := Temp1;                           Control := Temp1;
BODY:   Statement                           BODY:   Statement
        IF Control = Temp2 THEN GOTO EXIT           IF Control = Temp2 THEN GOTO EXIT
        Control := Control + 1                      Control := Control - 1
        GOTO BODY                                   GOTO BODY
EXIT:                                       EXIT:
```

These arrangements can be handled by the following parsing method

```
ForStatement<StackFrame frame>          (. int expType, count;
                                           boolean canChange;
                                           Label loopBody = new Label(!known);
                                           Label loopExit = new Label(!known);
                                           Entry var = new Entry();
                                           String name; .)
=  "for" Ident<out name>                (. var = Table.find(name);
                                           if (!var.declared)
                                             SemError("undeclared identifier");
                                           if (var.kind != Entry.Var)
                                             SemError("illegal control variable");
                                           if (!var.canChange)
                                             SemError("may not alter this variable");
                                           CodeGen.loadAddress(var);
                                           canChange = var.canChange;
                                           var.canChange = false;
                                           if (!isArith(var.type))
                                             SemError("control variable must be of arithmetic type"); .)
   ( "=" | ":="                           (. SemError("= intended?"); .)
   ) Expression<out expType>              (. if (!assignable(var.type, expType))
                                             SemError("incompatible with control variable");
                                           boolean up = true; .)
   ( "to" | "downto"                      (. up = false; .)
   ) Expression<out expType>              (. if (!assignable(var.type, expType))
                                             SemError("incompatible with control variable");
                                           CodeGen.startForLoop(up, loopExit);
                                           loopBody.here(); .)
   Statement<out count, frame, loopExit>
                                          (. if (count == 0 && warnings)
                                           Warning("empty statement part");
                                           CodeGen.endForLoop(up, loopBody);
                                           var.canChange = canChange;
                                           loopExit.here();
                                           CodeGen.pop(3); .) .
```

The code generation routines, as usual, are quite simple:

```
public static void startForLoop(boolean up, Label destination) {
// Generates prologue test for a for loop (either up or down)
  if (up) emit(PVM.sfu); else emit(PVM.sfd);
  emit(destination.address());
}

public static void endForLoop(boolean up, Label destination) {
// Generates epilogue test and increment/decrement for a for loop (either up or down)
  if (up) emit(PVM.efu); else emit(PVM.efd);
  emit(destination.address());
}

public static void pop(int n) {
// Generates code to discard top n elements from the stack
  emit(PVM.dsp); emit(-n);
}
```

but the magic that makes this work is contained in the interpreter with opcodes that are probably more complex than others you have seen to this point:

```
case PVM.sfu:           // start for loop "to"
  if (mem[cpu.sp + 1] > mem[cpu.sp]) cpu.pc = mem[cpu.pc];  // goto exit
  else {
    mem[mem[cpu.sp + 2]] = mem[cpu.sp + 1]; cpu.pc++;       // assign to control
  }
  break;
```

```
case PVM.sfd:           // start for loop "downto"          // goto exit
  if (mem[cpu.sp + 1] < mem[cpu.sp]) cpu.pc = mem[cpu.pc];
  else {
    mem[mem[cpu.sp + 2]] = mem[cpu.sp + 1]; cpu.pc++;       // assign to control
  }
  break;
case PVM.efu:           // end for loop "to"
  if (mem[mem[cpu.sp + 2]] == mem[cpu.sp]) cpu.pc++;        // loop complete
  else {
    mem[mem[cpu.sp + 2]]++; cpu.pc = mem[cpu.pc];           // increment control
  }
  break;
case PVM.efd:           // end for loop "downto"
  if (mem[mem[cpu.sp + 2]] == mem[cpu.sp]) cpu.pc++;        // loop complete
  else {
    mem[mem[cpu.sp + 2]]--; cpu.pc = mem[cpu.pc];           // decrement control
  }
  break;
```

Notes

- 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 the statement

```
for i = 35 to 64 write(i);   // loop to be executed 30 times
```

...	...	...	...	64	35	adr i	....

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

```
for i = initial to final statement
```

  which generates code

```
        LDA     i
        initial
        final
        SFU     L3
    L1  statement
    L2  EFU     L1
    L3  DSP     -3
```

- This solution allows for the introduction of guards againt accidental or deliberate corruption of the control variable within the *Statement* that forms the body of the loop. It requires a further change to the symbol table handler, which you can find in the solution kits.

- In Pascal the word `do` is also required, as illustrated below, and you were asked whether it would be a good idea to insist on it in Parva as well. Unfortunately the word `do` in the C family of languages is already bespoke for the *DoWhileStatement*, so if one were also to allow the latter it would not be a good idea!

```
ForStatement = "for" Ident "=" Expression ("to" | "downto") Expression "do" Statement .
```

Task 13 - 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;
*                                         boolean inc = true;
*                                         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");
*                                         if (!des.canChange)
*                                           SemError("may not alter this variable");
*                                         typeList.add(des.type); .)
        (
```

```
*          { "," Designator<out des>     (. if (des.entry.kind != Entry.Var)
*                                             SemError("invalid assignment");
*                                          if (!des.canChange)
*                                             SemError("may not alter this variable");
*                                          typeList.add(des.type);
*                                          desCount++; .)
*          }
           AssignOp
*          Expression<out expType>       (. if (count < typeList.size()
*                                              && !assignable(typeList.get(count++), expType))
*                                          SemError("incompatible types in assignment"); .)
*          { "," Expression<out expType> (. if (count < typeList.size()
*                                              && !assignable(typeList.get(count++), expType))
*                                          SemError("incompatible types in assignment");
*                                          expCount++; .)
*          }                             (. if (expCount != desCount)
*                                             SemError("left and right counts disagree");
*                                          CodeGen.assignN(desCount); .)
*       |   ( "++" | "-"                 (. inc = false; .)
*           )                            (. if (!isArith(des.type))
*                                             SemError("arithmetic type needed");
*                                          CodeGen.incOrDec(inc, des.type); .) .
        )
      | ( "++" | "--"                    (. inc = false; .)
        ) Designator<out des>            (. if (des.entry.kind != Entry.Var)
                                             SemError("variable designator required");
                                          if (!des.canChange)
                                             SemError("may not alter this variable");
                                          if (!isArith(des.type))
                                             SemError("arithmetic type required");
                                          if (des.isSimple)
                                             CodeGen.loadAddress(des.entry);
                                          CodeGen.incOrDec(inc, 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, but note the special check for the case where the is only one designator and one expression!

```
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.
  if (n > 1) { emit(PVM.ston); emit(n); }
  else emit(PVM.sto);
}
```

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 14 - Generating tighter PVM code

As stated in the prac sheet, this 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 canChange;    // false if entry is marked constant
*   public boolean isSimple;     // true unless it is an indexed designator
```

```
      public DesType(Entry entry) {
        this.entry = entry;
        this.type = entry.type;
*       this.canChange = entry.canChange;
*       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 != Entry.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 = Entry.noType;
                                             int size;
                                             DesType des;
                                             ConstRec con; .)
    =    Designator<out des>              (. type = des.type;
                                             switch (des.entry.kind) {
                                               case Entry.Var:
*                                                if (des.isSimple) CodeGen.loadValue(des.entry);
*                                                else CodeGen.dereference();
                                                 break;
                                               case Entry.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 = Entry.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 != Entry.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. It is all rather complex if we also wish to allow the Python multiple assignments! The solution below shows how we can achieve this by always using the `LDA, LDV` and `STO` opcodes for assignment. This might seem to defeat much of the advantage of using `LDL` and `STL`, but by clever factorisation of the grammar we can treat simple assignments with only one designator and one expression in a special way: Here was my initial solution:

```
   Assignment                          (. int op, expType;
                                          DesType des;
                                          boolean inc = true;
                                          int count = 0, desCount = 2, expCount = 1;
                                          ArrayList<Integer> typeList = new ArrayList<Integer>(); .)
   =  Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                            SemError("invalid assignment");
                                          if (!des.canChange)
                                            SemError("may not alter this variable");
                                          typeList.add(des.type); .)
       (  AssignOp
          Expression<out expType>      (. if (!assignable(des.type, expType))
                                            SemError("incompatible types in assignment");
*                                         if (des.isSimple)
*                                           CodeGen.storeValue(des.entry);
*                                         else CodeGen.assign(des.type); .)
*        |                              (. if (des.isSimple)
*                                           CodeGen.loadAddress(des.entry); .)
           ","  Designator<out des>     (. if (des.entry.kind != Entry.Var)
                                            SemError("invalid assignment");
                                          if (!des.canChange)
                                            SemError("may not alter this variable");
*                                         if (des.isSimple)
*                                           CodeGen.loadAddress(des.entry);
                                          typeList.add(des.type); .)
           { "," Designator<out des>    (. if (des.entry.kind != Entry.Var)
                                            SemError("invalid assignment");
                                          if (!des.canChange)
                                            SemError("may not alter this variable");
*                                         if (des.isSimple)
*                                           CodeGen.loadAddress(des.entry);
                                          typeList.add(des.type);
                                          desCount++; .)
           }
           AssignOp
           Expression<out expType>      (. if (count < typeList.size()
                                              && !assignable(typeList.get(count++), expType))
                                            SemError("incompatible types in assignment"); .)
           { ","
             Expression<out expType>    (. if (count < typeList.size()
                                              && !assignable(typeList.get(count++), expType))
                                            SemError("incompatible types in assignment");
                                          expCount++; .)
           }                            (. if (expCount != desCount)
                                            SemError("left and right counts disagree");
*                                         CodeGen.assignN(desCount); .)
         | ( "++" | "--"                (. inc = false; .)
           )                            (. if (!isArith(des.type))
                                            SemError("arithmetic type needed");
*                                         if (des.isSimple)
*                                           CodeGen.loadAddress(des.entry);
                                          CodeGen.incOrDec(inc, des.type); .)
       )
     |  ( "++" | "--"                   (. inc = false; .)
        ) Designator<out des>           (. if (des.entry.kind != Entry.Var)
                                            SemError("variable designator required");
                                          if (!des.canChange)
                                            SemError("may not alter this variable");
                                          if (!isArith(des.type))
                                            SemError("arithmetic type required");
*                                         if (des.isSimple)
*                                           CodeGen.loadAddress(des.entry);
                                          CodeGen.incOrDec(inc, des.type); .) .
```

The code generating routines needed are

```
public static void loadValue(Entry var) {
// Generates code to push value of variable var onto evaluation stack
  switch (var.offset) {
    case 0:  emit(PVM.ldl_0); break;
    case 1:  emit(PVM.ldl_1); break;
    case 2:  emit(PVM.ldl_2); break;
    case 3:  emit(PVM.ldl_3); break;
    default: emit(PVM.ldl); emit(var.offset); break;
  }
}

public static void storeValue(Entry var) {
// Generates code to pop top of stack and store at known offset.
  switch (var.offset) {
    case 0:  emit(PVM.stl_0); break;
    case 1:  emit(PVM.stl_1); break;
    case 2:  emit(PVM.stl_2); break;
    case 3:  emit(PVM.stl_3); break;
    default: emit(PVM.stl); emit(var.offset); break;
  }
}
```

You have heard me claim many times that "there is always a better way". And of course that is true. I am grateful to Michael Andersen, whose submission suggested how my solution above could be improved as shown below (the ideas were his, but they have been adapted to fit the rest of my code).

```
Assignment                          (. int op, expType;
                                       DesType des;
                                       boolean inc = true;
                                       int count = 0, desCount = 2, expCount = 1;
                                       ArrayList<DesType> desList = new ArrayList<DesType>(); .)
=  Designator<out des>               (. if (des.entry.kind != Entry.Var)
                                         SemError("invalid assignment");
                                       if (!des.canChange)
                                         SemError("may not alter this variable");
                                       desList.add(des); .)
     (  AssignOp
        Expression<out expType>       (. if (!assignable(des.type, expType))
                                         SemError("incompatible types in assignment");
                                       if (des.isSimple)
                                         CodeGen.storeValue(des.entry);
                                       else CodeGen.assign(des.type); .)
      | ","  Designator<out des>      (. if (des.entry.kind != Entry.Var)
                                         SemError("invalid assignment");
                                       if (!des.canChange)
                                         SemError("may not alter this variable");
                                       desList.add(des); .)
        { "," Designator<out des>     (. if (des.entry.kind != Entry.Var)
                                         SemError("invalid assignment");
                                       if (!des.canChange)
                                         SemError("may not alter this variable");
                                       desList.add(des);
                                       desCount++; .)
        }
        AssignOp
        Expression<out expType>       (. if (count < desList.size()
                                             && !assignable(desList.get(count++).type, expType))
                                         SemError("incompatible types in assignment"); .)
        { ","
          Expression<out expType>     (. if (count < desList.size()
                                             && !assignable(desList.get(count++).type, expType))
                                         SemError("incompatible types in assignment");
                                       expCount++; .)
        }                             (. if (expCount != desCount)
                                         SemError("left and right counts disagree");
                                       else {
                                         int sto = 0;
                                         while (expCount > 0) {
                                           expCount--;
                                           if (desList.get(expCount).isSimple)
                                             CodeGen.storeValue(desList.get(expCount).entry);
                                           else {
                                             CodeGen.assignOffset(expCount + sto);
                                             sto++;
                                           }
                                         }
                                         CodeGen.pop(sto);
                                       } .)
```

```
        | ( "++" | "--"                (. inc = false; .)
          )                            (. if (!isArith(des.type))
                                            SemError("arithmetic type needed");
                                          if (des.isSimple)
                                            CodeGen.loadAddress(des.entry);
                                          CodeGen.incOrDec(inc, des.type); .)
      )
    | ( "++" | "--"                    (. inc = false; .)
      ) Designator<out des>            (. if (des.entry.kind != Entry.Var)
                                            SemError("variable designator required");
                                          if (!des.canChange)
                                            SemError("may not alter this variable");
                                          if (!isArith(des.type))
                                            SemError("arithmetic type required");
                                          if (des.isSimple)
                                            CodeGen.loadAddress(des.entry);
                                          CodeGen.incOrDec(inc, des.type); .) .
```

We need a new code generating method:

```
public static void assignOffset(int offset) {
// Generates code to store value currently on top-of-stack on the address
// given by the element<offset> below the TOS.
   emit(PVM.stof); emit(offset);
}
```

and a new clause in the interpreter.

```
  case PVM.stof:          // store a value from top of stack to address an offset from sp
    int offset = next();
    if (offset < 0) ps = badAdr;
    else if (inBounds(cpu.sp + offset + 1)) mem[mem[cpu.sp + offset + 1 ]] = pop();
    break;
```

Just for further interest, the full solution in the solution kit allows the user to choose between "optimized" and "regular" old-style code by using a pragma `$O+` or command line option `-o`.