Computer Science 3 - 2010

Programming Language Translation

Practical for Weeks 25 - 26, beginning 11 October 2010 - solutions

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

Task 2 - Better use of the debugging pragma

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

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

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

```
   PRAGMAS
*    CodeOn       = "$C+" .              (. listCode  = true; .)
*    CodeOff      = "$C-" .              (. listCode  = false; .)
     DebugOn      = "$D+" .              (. debug = true; .)
     DebugOff     = "$D-" .              (. debug = false; .)
*    ShortCircOn  = "$S+" .              (. shortCirc = true; .)
*    ShortCircOff = "$S-" .              (. shortCirc = 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 if (args[i].toLowerCase().equals("-s")) Parser.shortCirc = false;
     else inputName = args[i];
   }
   if (inputName == null) {
     System.err.println("No input file specified");
*    System.err.println("Usage: Parva [-l] [-d] [-w] [-c] [-s] source.pav [-l] [-d] [-w] [-c] [-s]");
     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.err.println("-s turn off short-circuit evaluation");
     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 - More variety in identifier names

The problem called for a trivial change to the definition of the identifier token so as to prevent an identifier from having an underscore as its final character:

```
   TOKENS
*     identifier   = letter { letter | digit | "_" { "_" } ( letter | digit ) } .
// or identifier   = letter { { "_" } (letter | digit ) } .
```

Task 4 - 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):

```
   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:

```
   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 5 - The final word in declarations

The problem called for the introduction of the `final` keyword in variable declarations, so as to allow code of the form:

```
final int max = 2;
int i = 5;
final int iPlusMax = i + max;
final int[] list = new int[iPlusMax];
```

The key to this is to add an extra field to those in the `Entry` class used by the table handling routines:

```
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,
    voidType =  8;

  public int     kind;
  public String  name;
  public int     type;
  public int     value;       // constants
  public int     offset;      // variables
  public Entry   nextInScope; // link to next entry in current scope
  public boolean declared;    // true for all except sentinel entry
*   public boolean canChange;   // true except for constants
} // end Entry
```

A similar addition is needed in the `DesType` class

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

With these in place the parsers for handling *VarDeclarations* can set the fields correctly:

```
   VarDeclarations<StackFrame frame>   (. int type;
                                          boolean canChange = true; .)
*  =  [ "final"                        (. canChange = false; .)
*     ] Type<out type>
```

```
*     OneVar<frame, type, canChange>
*     { WEAK "," OneVar<frame, type, canChange> }
      WEAK ";" .

   OneVar<StackFrame frame, int type, boolean canChange>
                                       (. int expType;
                                          Entry var = new Entry(); .)
   =  Ident<out var.name>              (. var.kind = Entry.Var;
                                          var.type = type;
*                                         var.canChange = canChange;
                                          var.offset = frame.size;
                                          frame.size++; .)
      (   AssignOp                     (. CodeGen.loadAddress(var); .)
          Expression<out expType>      (. if (!compatible(var.type, expType))
                                            SemError("incompatible types in assignment");
                                          CodeGen.assign(var.type); .)
*       |                              (. if (!canChange)
*                                            SemError("defining expression required"); .)
      )                                (. Table.insert(var); .) .
```

where, it should be noted, it is an error to apply `final` to a variable declaration if the initial definition of a value for that variable is omitted. The `canChange` field is then checked at the points where one might attempt to alter a variable marked as `final`, namely within *AssignmentStatements* and *ReadElements*:

```
   Assignment                          (. int expType;
                                          DesType des; .)
   =  Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                            SemError("invalid assignment");
*                                         if (!des.canChange)
*                                           SemError("may not alter this variable");
      AssignOp
      Expression<out expType>          (. if (!compatible(des.type, expType))
                                            SemError("incompatible types in assignment");
                                          CodeGen.assign(des.type); .) .

   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:
                                              CodeGen.read(des.type); break;
                                            default:
                                              SemError("cannot read this type"); break;
                                          } .) .
```

There is one further subtlety. Marking an "array" as `final` implies only that the *reference* to the array may not be altered, not that individual elements may not be altered. This necessitates a tweak to the *Designator* parser:

```
   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 (entry.kind == Entry.Var) CodeGen.loadAddress(entry); .)
*     [  "["                          (. des.canChange = true;
                                          if (isRef(des.type)) des.type--;
                                          else SemError("unexpected subscript");
                                          if (entry.kind != Entry.Var)
                                            SemError("unexpected subscript");
                                          CodeGen.dereference(); .)
            Expression<out indexType>  (. if (!isArith(indexType))
                                            SemError("invalid subscript type");
                                          CodeGen.index(); .)
        "]"
      ] .
```

Task 6 - Better string handling

This is easier than might at first appear. The code generation routine for the new opcode becomes

```
public static void writeStr(String str) {
// Generates code to output str after storing it in the string pool
  emit(PVM.prnsp); emit(PVM.addString(str));
}
```

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

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

public static int addString(String str) {
// Adds str to string pool if not already there, returns its index
  int i = 0;
  for (i = 0; i < strPool.size(); i++)
    if (str.equals(strPool.get(i))) return i;
  /* if not found */ strPool.add(str); return i;
}
```

The interpretation of the `PVM.prnsp` opcode has to alter:

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

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

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

Task 7 - 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. 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 conversion; the `(char)` cast also introduces the generation of 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 8 - What if you don't like short-circuit evaluation of Boolean expressions?

This is easily implemented as follows, where we have also shown how the feature is controlled by the 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 = Entry.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 = Entry.boolType; .)
      }                                   (. shortcircuit.here(); .) .
```

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

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

```
   Assignment                            (. int expType;
                                            DesType des;
*                                           boolean inc = true; .)
   =    Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                               SemError("invalid assignment");
                                            if (!des.canChange)
                                               SemError("may not alter this variable"); .)
        (   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 (!des.canChange)
*                                               SemError("may not alter this variable");
*                                           if (!isArith(des.type))
*                                               SemError("arithmetic type needed");
*                                           CodeGen.incOrDec(inc, des.type); .) .
```

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 10 - This has gone on long enough - time for a break

The syntax of the *BreakStatement* is, of course, trivial. The catch is that one has to allow these statements only in the context of loops. To find a context-free grammar with this restriction is not worth the effort.

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>
             | ForStatement<frame>
             | SwitchStatement<frame>
*            | BreakStatement<breakLabel>
             | HaltStatement
             | ReturnStatement
             | ReadStatement
             | WriteStatement
             | ";"
             | "stackdump" ";"          (. if (debug) CodeGen.dump(); .)
          ) .

*  Block<StackFrame frame, Label breakLabel>
   =                                    (. Table.openScope(); .)
*     "{" { Statement<frame, breakLabel>
         } 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); .)
   =  "if" "(" Condition ")"            (. CodeGen.branchFalse(falseLabel); .)
      [ "then"                          (. SemError("then is not used in Parva"); .)
*     ] Statement<frame, breakLabel>    (. falseLabel.here(); .) .

   WhileStatement<StackFrame frame>     (. Label loopStart = new Label(known); .)
   =  "while" "(" Condition ")"         (. Label loopExit = new Label(!known);
                                           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 ";" .
```

Task 11 - 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. The exercises suggested implementing each of two possible forms of *ForStatement*.

The full solution, by carefully factoring the grammar, shows how both forms might be possible within one language definition. For the purposes of discussion it is convenient to treat each separately.

A possible syntax for a *ForStatement*, suggested by the C family of languages, might be:

```
ForStatement = "for" ([ [BasicType] Ident "=" Expression ] ";" [ Condition ] ";" [ Assignment] ) Statement .
```

The semantics of this statement are deemed to be essentially equivalent to

```
Ident "=" Expression;
while ( Condition ) {
  Statement;
  Assignment;
}
```

The solution follows:

```
ForStatement<StackFrame frame>        (. int type = Entry.noType, expType;
                                         boolean local = false;
                                         boolean canChange;
                                         Label loopExit = new Label(!known);
                                         Label loopContinue = new Label(!known);
                                         Label loopBody = new Label(!known);
                                         Entry var = new Entry();
                                         String name; .)
=  "for"
      "("                             (. Table.openScope(); .)
         [ [ BasicType<out type>      (. local = true; .)
           ]
           Ident<out name>            (. if (local) {
                                           var.name = name;
                                           var.kind = Entry.Var;
                                           var.type = type;
                                           var.offset = frame.size;
                                           var.canChange = true;
                                           frame.size++;
                                          }
                                          else {
                                            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");
                                          }
                                         if (!isArith(var.type))
                                           SemError("illegal type of control variable"); .)
           ( "=" | ":="                 (. SemError("= intended?"); .)
           )                            (. CodeGen.loadAddress(var); .)
           Expression<out expType>      (. if (!assignable(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.assign(var.type);
                                         if (local) Table.insert(var); .)
         ] ";"                          (. Label testLabel = new Label(known); .)
         (   Condition
           |                            (. CodeGen.loadConstant(1); .)
         )                              (. CodeGen.branchFalse(loopExit);
                                           CodeGen.branch(loopBody); .)
         ";"                            (. loopContinue.here(); .)
         [ Assignment ]                 (. CodeGen.branch(testLabel); .)
      ")"                               (. loopBody.here();
                                           canChange = var.canChange;
                                           var.canChange = false; .)
      Statement<frame, loopExit>
                                        (. CodeGen.branch(loopContinue);
                                           loopExit.here();
                                           var.canChange = canChange;
                                           Table.closeScope(); .)
```

Notes

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

  A statement like

```
for ( i = initial ; condition; assignment ) statement
```

  generates the code

```
      LDA     i
      initial
      STO
L1    condition
      BZE     L4    (exit)
      BRN     L3    (body)
L2    assignment
      BRN     L1    (condition)
L3    statement
      BRN     L2    (assignment)
L4    ...
```

- The value of the control variable should not be changed by the *Statement* forming the loop body. Before this statement is parsed, the `canChange` status of the control variable is saved, and then altered to `false`, which will cause any attempts to change it to be flagged as errors. After the loop body has been parsed, the status is reset. Preservation is necessary in case the *ForStatement* itself forms part of the body of an outer *ForStatement* or *Block* in which the control variable is already protected (perhaps by virtue of being marked `final`).

Task 12 - For what it's worth, Pascal did it differently

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

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

These arrangements can be handled by the following parsing method

```
ForStatement<StackFrame frame>        (. int expType;
                                         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<frame, loopExit>      (. 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
```

- One again, this solution guards againt accidental or deliberate corruption of the control variable within the *Statement* that forms the body of the loop.

- 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 - Beg, borrow and steal ideas from other languages

This exercise called on you to extend Parva to adopt an idea used in Pascal, where a statement like

```
write(X : 5, X + A : 12, X - Y : 2 * N);
```

will write the values of `X, X+A` and `X-Y` in fields of widths `5, 12` and `2*N` respectively. This is easily handled by modifying the production for *WriteElement*:

```
WriteElement                          (. int expType, formType;
                                         boolean formatted = false;
                                         String str; .)
=   StringConst<out str>              (. CodeGen.writeStr(str); .)
  | Expression<out expType>           (. if (!(isArith(expType) || expType == Entry.boolType))
                                           SemError("cannot write this type"); .)
*     [ ":"  Expression<out formType> (. if (formType != Entry.intType)
*                                        SemError("fieldwidth must be integral");
*                                       formatted = true; .)
*     ]                               (. switch (expType) {
                                           case Entry.intType:
                                           case Entry.boolType:
                                           case Entry.charType:
*                                            CodeGen.write(expType, formatted); break;
                                           default:
                                             break;
                                         } .) .
```

and modifying the code generation routine

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

and new opcodes whose interpretation is

```
case PVM.prnfi:         // integer output formatted
  if (tracing) results.write(padding);
  fieldWidth = pop();
  results.write(pop(), fieldWidth);
  if (tracing) results.writeLine();
  break;
case PVM.prnfb:         // boolean output formatted
  if (tracing) results.write(padding);
  fieldWidth = pop();
  if (pop() != 0) results.write(" true  ", fieldWidth);
  else results.write(" false ", fieldWidth);
  if (tracing) results.writeLine();
  break;
case PVM.prnfc:         // character output formatted
  if (tracing) results.write(padding);
  fieldWidth = pop();
  results.write((char) (Math.abs(pop()) % (maxChar + 1)), fieldWidth);
  if (tracing) results.writeLine();
  break;
```

Task 14 - Switch to Parva - satisfaction guaranteed! (Big Bonus if you get this far.)

The exercise suggested an 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 can duplicate the value on the top of the stack 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. This calls for yet another simple opcode for the PVM, which can be generated by calling:

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

and whose 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;
```

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. To effect the test for membership of a list of labels like the -9, 9, -10 and 10 of the examples we play the same game again - another code generation routine:

```
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;
```

The code for the parsing routine follows:

```
SwitchStatement<StackFrame frame>     (. int expType, 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 == Entry.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 != Entry.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. These 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.

- 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
        ...
```