Computer Science 3 - 2016

Programming Language Translation

Practical 7 - Week beginning 5 September 2016 - solutions

Sources of full solutions for these problems may be found on the course web page as the file `PRAC7A.ZIP`.

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:

The definitions of the pragmas which will modify these flags follow:

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

**   StackDump   = "$SD" .                (. if (debug) CodeGen.Stack(); .)
**   HeapDump    = "$HD" .                (. if (debug) CodeGen.Heap(); .)
**   TableDump   = "$ST" .                (. if (debug) Table.PrintTable(OutFile.StdOut); .)
```

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].ToLower().Equals("-l")) mergeErrors = true;
       else if (args[i].ToLower().Equals("-d")) Parser.debug = true;
**     else if (args[i].ToLower().Equals("-c")) Parser.listCode = true;
       else if (args[i].ToLower().Equals("-n")) execution = false;
       else if (args[i].ToLower().Equals("-g")) immediate = true;
       else inputName = args[i];
     }
     if (inputName == null) {
       Console.WriteLine("No input file specified");
       Console.WriteLine("Usage: Parva [-l] [-d] [-c] [-n] [-g] source.pav");
       Console.WriteLine("-l directs source listing to listing.txt");
       Console.WriteLine("-d turns on debug mode");
**     Console.WriteLine("-c lists object code (.cod file)");
       Console.WriteLine("-n no execution after compilation");
       Console.WriteLine("-g execute immediately after compilation (StdIn/StdOut)");
       System.Environment.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 the production 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 - Two of the three Rs - Reading and Writing

The extensions to *ReadStatement* and *WriteStatement* are very simple. It is useful to allow both the `readLine()` and `writeLine()` forms of these statements to have empty argument lists, something a little meaningless for the simple `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!). And while we are at it, let's make the *HaltStatement* as general as possible.

```
**  ReadStatement
**  = (  "read"     "("   ReadList    ")"
**     | "readLine" "(" [ ReadList ]  ")"   (.CodeGen.ReadLine(); .)
**    )
**    WEAK ";" .
**
**  ReadList
**  = ReadElement { WEAK "," ReadElement } .
**
**  WriteStatement
**  = (   "write"     "("   WriteList   ")"
**      | "writeLine" "(" [ WriteList ] ")" (. CodeGen.WriteLine(); .)
**    )
**    WEAK ";" .
**
**  WriteList
**  = WriteElement { WEAK "," WriteElement } .
**
**  HaltStatement                          /* optional arguments! */
**  = "halt" [ "(" [ WriteList ] ")" ]
**    WEAK ";"                             (. CodeGen.LeaveProgram(); .) .
**
```

Task 5 - Repeat discussions with your team mates until you all get the idea

They don't get much easier than this.

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

For the *DoWhileStatement* we similarly need a single label, and a single conditional branch that goes to the start of the loop body. The only trick is that we don't have a "branch on true" opcode - but all we have to do is to generate a "negate boolean" operation that will be applied to the computed value of the *Condition* at run time before the conditional branch takes effect:

```
DoWhileStatement<StackFrame frame>     (. Label loopStart = new Label(known); .)
=  "do"
     Statement<frame>
   WEAK "while"
   "(" Condition ")" WEAK ";"          (. CodeGen.NegateBoolean();
                                          CodeGen.BranchFalse(loopStart); .) .
```

Task 6 - 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)))
```

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 these by a call to a new code generating function for each *Expression* after the first. Convince yourselves of the logic behind the auto-promotion to integer type if *any* of the arguments are of the integer type.

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

The code generator and the PVM are easily extended

```
public static void MaxMin(bool max) {
// Generates code to leave max/min(tos, sos) on top of stack
  Emit(max ? PVM.max : PVM.min);
} // CodeGen.MaxMin

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 solution kit.

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

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

along with a code generator method as follows:

```
public static void MaxMin(bool 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);
} // CodeGen.MaxMin
```

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 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* efficiently 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. This is a very useful technique to remember.

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

Many - perhaps most - people in attempting this problem come up with the following sort of code instead. This can generate `BRN` instructions where none are needed.

```
   IfStatement<StackFrame frame>          (. Label falseLabel = new Label(!known);
                                             Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"              (. CodeGen.BranchFalse(falseLabel); .)
      [ "then"                            (. SemError("then is not used in Parva"); .)
      ]  Statement<frame>                 (. CodeGen.Branch(outLabel);
                                             falseLabel.Here(); .)
      [ "else"  Statement<frame>  ]       (. outLabel.Here(); .) .
```

Using this strategy, source code like

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

would lead 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    ....
```

Although you were not asked for it, we can handle *elsif* clauses with the same idea. Note that. after defining `falseLabel.Here()`, the label is "re-used" by assigning it another instance of an "unknown" label. If you don't do this you will get all sorts of bad code or funny messages from the label handler!

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

Task 8 - This has gone on long enough - time for a break (and then continue)

Although I asked only for the *break* statement, we may as well illustrate the less common *continue* statement as well, for those who like reading interesting, clear, code in a compiler…

The syntax of the *BreakStatement* and *ContinueStatement* 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.

A common approach to incorporating context-sensitive checking in conjunction with code generation, as hopefully you know, is based on passing information as parameters between subparsers. The pieces of information we need to pass here are `Label` objects. We change the parser for *Statement* and for *Block* as follows:

```
**  Block<StackFrame frame, Label breakLabel, Label continueLabel>
    =                                   (. Table.openScope(); .)
**     "{" { Statement<frame, breakLabel, continueLabel>
           }
        WEAK "}"                        (. Table.closeScope(); .) .

**  Statement<StackFrame frame, Label breakLabel, Label continueLabel>
**  =  SYNC (   Block<frame, breakLabel, continueLabel>
             | ConstDeclarations
             | VarDeclarations<frame>
             | AssignmentOrCall
             | IncDecStatement
**           | IfStatement<frame, breakLabel, continueLabel>
             | WhileStatement<frame>
             | DoWhileStatement<frame>
             | RepeatStatement<frame>
**           | BreakStatement<breakLabel>
**           | ContinueStatement<continueLabel>
             | HaltStatement
             | ReturnStatement
             | ReadStatement
             | WriteStatement
             | ";"                      (. if (warnings) Warning("empty statement"); .)
          ) .
```

The very first call to *Statement* passes `null` as the value for each of these labels:

```
    Body<StackFrame frame>              (. Label DSPLabel = new Label(known);
                                           int sizeMark = frame.size;
                                           CodeGen.OpenStackFrame(0); .)
**  =  "{" { Statement<frame, null, null> }
       WEAK "}"                         (. CodeGen.FixDSP(DSPLabel.Address(), frame.size - sizeMark);
                                           CodeGen.LeaveVoidFunction(); .) .
```

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

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

   WhileStatement<StackFrame frame>     (. Label loopExit  = new Label(!known);
**                                         Label loopContinue = new Label(known); .)
   =  "while" "(" Condition ")"         (. CodeGen.BranchFalse(loopExit); .)
**    Statement<frame, loopExit, loopContinue>
                                        (. CodeGen.Branch(loopContinue);
**                                         loopExit.Here(); .) .
```

```
** DoWhileStatement<StackFrame frame>      (. Label loopExit = new Label(!known);
**                                           Label loopContinue = new Label(known);
                                            Label loopStart = new Label(known); .)
   =  "do"
**      Statement<frame, loopExit, loopContinue>
**    WEAK "while"                          (. loopContinue.Here(); .)
      "(" Condition ")" WEAK ";"            (. CodeGen.NegateBoolean();
                                            CodeGen.BranchFalse(loopStart);
**                                          loopExit.Here(); .) .

** RepeatStatement<StackFrame frame>       (. Label loopExit = new Label(!known);
**                                           Label loopContinue = new Label(!known);
                                            Label loopStart = new Label(known); .)
   =  "repeat" {
**      Statement<frame, loopExit, loopContinue>
        }
      WEAK "until"                          (. loopContinue.Here(); .)
      "(" Condition ")" WEAK ";"            (. CodeGen.BranchFalse(loopStart);
**                                          loopExit.Here(); .) .

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

   ContinueStatement<Label continueLabel>
** =  "continue"                           (. if (continueLabel == null)
**                                             SemError("continue is not allowed here");
**                                          else CodeGen.Branch(continueLabel); .)
      WEAK ";" .
```

There is at least one other way of solving the problem, which involves using local variables in the parsing methods to "stack" up the old labels, assigning new ones, and then restoring the old ones afterwards. But the method just presented seems the neatest.

Task 9 - Make the change; enjoy life; upgrade now to Parva++

It might not at first have been obvious, but I fondly hope that everyone will realize that this extension is handled at the initial level by further modifications to the *AssignmentOrCall* production, which used a conflict resolver to avoid LL(1) conflicts. The code below handles these tasks (including the tests for assignment 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.

```
    AssignmentOrCall                    (. int expType;
                                           DesType des;
                                           bool inc = true; .)
    =  (  IF (IsCall(out des))          // use resolver to handle LL(1) conflict
          identifier
          "("                           (. CodeGen.FrameHeader(); .)
             Arguments<des>             (. CodeGen.Call(des.entry.entryPoint); .)
          ")"
       |  Designator<out des>           (. if (des.entry.kind != Kinds.Var)
                                             SemError("cannot assign to " + Kinds.kindNames[des.entry.kind]); .
**        (   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); .)
**        )
       ) WEAK ";" .

**  IncDecStatement                     (. DesType des;
**                                         bool inc = true; .)
**  =  ( "++" | "--"                    (. inc = false; .)
**     ) Designator<out des>            (. if (des.entry.kind != Kinds.Var)
**                                          SemError("variable designator required");
**                                       if (!IsArith(des.type))
**                                          SemError("arithmetic type needed");
**                                       CodeGen.IncOrDec(inc, des.type); .)
**     WEAK ";" .
```

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

```
   public static void IncOrDec(bool inc, int type) {
   // Generates code to increment the value found at the address currently
   // stored at the top of the stack.
   // If necessary, apply character range check
**   if (type == Types.charType) Emit(inc ? PVM.incc : PVM.decc);
**   else Emit(inc ? PVM.inc : PVM.dec);
   } // CodeGen.IncOrDec
```

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:           // integer ++
  adr = Pop();
  if (InBounds(adr)) mem[adr]++;
  break;

case PVM.dec:           // integer --
  adr = Pop();
  if (InBounds(adr)) mem[adr]--;
  break;

case PVM.incc:          // character ++ (checked)
  adr = Pop();
  if (InBounds(adr))
    if (mem[adr] < maxChar) mem[adr]++;
    else ps = badVal;
  break;

case PVM.decc:          // character -- (checked)
  adr = Pop();
  if (InBounds(adr))
    if (mem[adr] > 0) mem[adr]--;
    else ps = badVal;
  break;
```

Task 10 - The array is the object of the exercise

Since the heap allocator stores the length of an array on the heap below the space for the array itself, all we need to do is dereference the address of the array (to find the address of its position on the heap), and then dereference this address to find the length of the array.

We need to provide another option within *Primary* and indulge in the usual spate of semantic checking:

```
| "Length" "(" Expression<out type2>   (. if (!IsArray(type2))
                                           SemError("not an object"); .)
  ")"                                  (. CodeGen.Dereference();
                                          type = Types.intType; .)
```

This is a very simple addition. Will the students who claim that Pat Terry's Compiler Course is *really difficult* please stop spreading such rumours?

Notice that we have expressed the parameter to the *Length* function as an *Expression* and not as a *Designator*. This would allow us to write code like `Length(Copy(array))`, which the "obvious" *Designator* would forbid.

Syntactically, the extensions to copy an array and to see whether two arrays are equal element-wise are also straightforward. Two more options within *Primary* and a couple of new code generating calls do the trick:

```
| "Equals" "(" Expression<out type>    (. if (!IsArray(type))
                                           SemError("not an object"); .)
  "," Expression<out type2>            (. if (!IsArray(type2))
                                           SemError("not an object");
                                          if (!Compatible(type, type2))
                                           SemError("incomparable operand types");
                                          CodeGen.ObjectEquals();
                                          type = Types.boolType; .)
  ")"
```

```
  | "Copy" "(" Expression<out type>      (. if (!IsArray(type))
                                               SemError("not an object"); .)
    ")"                                (. CodeGen.Copy(); .)
```

I repeat: Will the students who claim that Pat Terry's Compiler Course is *really difficult* please stop spreading such rumours?

Oh well, yes, the extensions to the PVM take a bit of thought. The evaluation of the expressions compiled from the above productions will have left the address of the base of the array on the top of the stack, so we know where to find the victims, and a simple dereference (built into the code below) finds the limit needed for the loop that will process the elements of the array, as below. Surely *not really* difficult.

Once again we see how the ability to "redefine" our machine makes for very easy compilation, at the expense of some slight inefficiency on the part of the interpreter.

```
case PVM.equal:         // test two arrays for deep equality
  tos = Pop();
  sos = Pop();
  if (InBounds(tos) && InBounds(sos)) { // check null references
    int length1 = mem[tos];             // dereferencing gives lengths
    int length2 = mem[sos];
    if (length1 != length2)             // must have the same length!
      Push(0);
    else {
      bool match = true;
      loop = 0;
      while (match && loop <= length1) {
        if (mem[tos + loop] != mem[sos + loop]) match = false;
        loop++;
      }
      Push(match ? 1 : 0);
    }
  }
  break;

case PVM.copy:          // clone heap array allocation
  int old = Pop();
  if (InBounds(old)) {                  // check null reference
    int length = mem[old];              // dereference gives the length
    if (length <= 0 || length + 1 > cpu.sp - cpu.hp - 2)
      ps = badAll;                      // heap is full
    else {
      for (loop = 0; loop <= length; loop++) // copy source elements
        mem[cpu.hp + loop] = mem[old + loop];
      Push(cpu.hp);                     // stack the address of the new array
      cpu.hp += length + 1;             // bump heap pointer
    }
  }
  break;
```

Task 11 - 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 {
// All fields initialized, but are modified after construction (by semantic analyser)
  public int     kind         = Kinds.Var;
  public string  name         = "";
  public int     type         = Types.noType;
  public int     value        = 0;       // constants
  public int     offset       = 0;       // variables
  public bool    declared     = true;    // true for all except sentinel entry
  public Entry   nextInScope  = null;    // link to next entry in current scope
```

```
     public int     nParams      = 0;       // functions
     public Label   entryPoint   = new Label(false);
     public Entry   firstParam   = null;
**   public bool    cannotAlter  = false;   // true if value cannot be changed
   } // 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 bool cannotAlter;     // false if entry is marked as immutable

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

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

```
   VarDeclarations<StackFrame frame>       (. int type;
                                              bool cannotAlter = false; .)
** =  [ "final"                            (. cannotAlter = true; .)
**    ] Type<out type>
**    OneVar<frame, type, cannotAlter>
**    { WEAK "," OneVar<frame, type, cannotAlter> }
      WEAK ";" .

   OneVar<StackFrame frame, int type, bool cannotAlter>
                                           (. int expType;
                                              Entry var = new Entry(); .)
   =  Ident<out var.name>                  (. var.kind = Kinds.Var;
                                              var.type = type;
**                                            var.cannotAlter = cannotAlter;
                                              var.offset = frame.size;
                                              frame.size++; .)
      (   AssignOp                         (. CodeGen.LoadAddress(var); .)
          Expression<out expType>          (. if (!Assignable(var.type, expType))
                                                SemError("incompatible types in assignment");
                                              CodeGen.Assign(var.type); .)
**      |                                  (. if (cannotAlter)
**                                               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 `cannotAlter` field is then checked at the points where one might attempt to alter a variable marked as `final`, namely within *AssignmentStatements* and *ReadElements*:

```
   AssignmentOrCall                        (. int expType;
                                              DesType des;
                                              bool inc = true; .)
   =  (  IF (IsCall(out des))              // use resolver to handle LL(1) conflict
         identifier
         "("                               (. CodeGen.FrameHeader(); .)
            Arguments<des>                 (. CodeGen.Call(des.entry.entryPoint); .)
         ")"
      |  Designator<out des>               (. if (des.entry.kind != Kinds.Var)
                                                SemError("cannot assign to " + Kinds.kindNames[des.entry.kind]);
**                                            if (des.cannotAlter)
**                                              SemError("you 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); .)
         )
      ) WEAK ";" .
```

```
   IncDecStatement                          (. DesType des;
                                               bool inc = true; .)
   =  ( "++" | "--"                         (. inc = false; .)
      ) Designator<out des>                 (. if (des.entry.kind != Kinds.Var)
                                                SemError("variable designator required");
**                                            if (des.cannotAlter)
**                                              SemError("you may not alter this variable");
                                              if (!IsArith(des.type))
                                                SemError("arithmetic type needed");
                                              CodeGen.IncOrDec(inc, des.type); .)
      WEAK ";" .
   .

   ReadElement                              (. String str;
                                               DesType des; .)
   =   StringConst<out str>                 (. CodeGen.WriteStr(str); .)
     | Designator<out des>                  (. if (des.entry.kind != Kinds.Var)
                                                 SemError("wrong kind of identifier");
**                                            if (des.cannotAlter)
**                                              SemError("you 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 == Kinds.Var) CodeGen.LoadAddress(entry); .)
**    [  "["                                (. des.cannotAlter = false;
                                               if (IsArray(des.type)) des.type--;
                                               else SemError("unexpected subscript");
                                               if (entry.kind != Kinds.Var)
                                                 SemError("unexpected subscript");
                                               CodeGen.Dereference(); .)
           Expression<out indexType>        (. if (!IsArith(indexType))
                                                 SemError("invalid subscript type");
                                               CodeGen.Index(); .)
        "]"
      ] .
```

Task 12 - 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;
                                               bool formatted = false;
                                               string str; .)
   =   StringConst<out str>                 (. CodeGen.WriteString(str); .)
     | Expression<out expType>              (. if (!(IsArith(expType) || expType == Types.boolType))
                                                 SemError("cannot write this type"); .)
**     [ ":"  Expression<out formType>       (. if (formType != Types.intType)
**                                              SemError("fieldwidth must be integral");
**                                            formatted = true; .)
**     ]                                    (. switch (expType) {
                                                 case Types.intType:
                                                 case Types.boolType:
                                                 case Types.charType:
**                                                 CodeGen.Write(expType, formatted); break;
                                                 default:
                                                   break;
                                               } .) .
```

and modifying the code generation routine

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

and adding new opcodes whose interpretation is

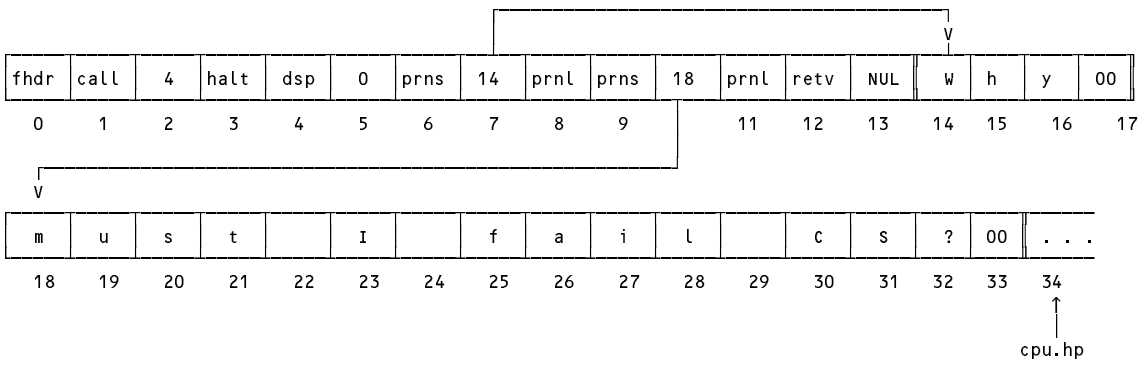
```
case PVM.prnfi:         // integer output formatted
  if (tracing) results.Write(padding);
  fieldWidth = Pop();
  results.Write(Pop(), fieldWidth);
  if (tracing) results.WriteLine();
  break;

case PVM.prnfb:         // bool 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 13 - Let's untangle that heap of string

Not many people tried this. The group who did made a brave but incorrect attempt.

Until now, any strings encountered in a Parva program have been stacked at the top end of memory, whence they hav been extracted by the PRNS opcode in the PVM.

In an alternative model, one that is used in the PAM (Parva Actual Machine), the strings are stored above the program code - effectively in space that the PVM would otherwise have used for its heap - like this:



for the very simple program

```
void main () {
// A fatalistic approach to the oncoming examinations?
  writeLine("Why");
  writeLine("must I fail CS?");
}
```

It was suggested that one might associate a label with each PRNS operation, since at the time one wants to generate the opcode it is not known exactly where the string will be stored. Furthermore, to develop tight code suited to a small machine like the PAM, one should optimize the analysis and code generation so that if the same string appears in the program in two or more places, only one copy is loaded in memory.

To do this it is expedient to define a small class whose instances can store a string and a list of references (labels) to one or more incomplete PRNS operations:

```
**   class StringEntry {
**     public Label reference;
**     public string str;
**
**     public StringEntry(string str) {
**       this.reference = new Label(false);
**       this.str = str;
**     }
**   } // end StringEntry
```

Then `WriteString()`, the code generator invoked when a *ReadElement* or *WriteElement* is parsed becomes

```
**    public static void WriteString(string str) {
**    // Generates code to output string stored at known location in program memory
**      int i = 0;
**      StringEntry entry = new StringEntry(str);
**      while (i < stringList.Count && !str.equals(stringList[i].str)) i++;  // search first
**      if (i >= stringList.Count)
**        stringList.Add(entry);
**      else
**        entry = stringList[i];
**      Emit(PVM.prns); Emit(entry.reference.address());
**    } // CodeGen.WriteString
```

At the end of compilation this list is scanned:

```
   Parva                                (. CodeGen.FrameHeader();         // no arguments
                                           CodeGen.Call(mainEntryPoint);  // forward, incomplete
                                           CodeGen.LeaveProgram(); .)     // return to O/S
   = { FuncDeclaration } EOF            (. if (!mainEntryPoint.IsDefined())
                                             SemError("missing Main function");
*                                       CodeGen.FixStrings(); .) .
```

Each forward reference is resolved (the PRNS instructions are "backpatched"), and the corresponding string is copied, character by character, to the code array. Note that the end of the executable code is first marked with the fictitious high-numbered opcode `PVM.nul` to allow `PVM.ListCode()` to distinguish code from strings.

```
**   public static void FixStrings() {
**     Emit(PVM.nul);                          // mark end of real code
**     foreach (StringEntry s in stringList) {
**       BackPatch(s.reference.Address());     // fix one or more PRNS instructions
**       for (int i = 0; i < s.str.Length; i++)
**         Emit(s.str[i]);                     // copy character by character
**       Emit(0);                              // null-terminated
**     }
**   } // CodeGen.FixStrings
```

The interpreter executes a `PRNS` opcode with a loop, as before, save that this time it runs "forward".

```
   case PVM.prns:          // string output
     if (tracing) results.WriteLine();
     loop = Next();
     while (ps == running && mem[loop] != 0) {
**     results.Write((char) mem[loop]); loop++;
**     if (loop > heapBase) ps = badMem;
     }
     if (tracing) results.WriteLine();
     break;
```

and this change is needed in the `listCode()` method as well:

```
        case PVM.prns:
          i = (i + 1) % memSize;
          j = mem[i]; codeFile.Write(" \"");
          while (mem[j] != 0) {
            switch (mem[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((char) mem[j]); break;
            }
    **      j++;
          }
          codeFile.Write("\"");
          break;
```

Task 14 - In case you have nothing better to do (Switch to Parva - Success Guaranteed)

Nobody rose to this challenge. The problem called for the implementation of a *SwitchStatement* described by the productions:

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

as exemplified by

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

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

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

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

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

```
SwitchStatement<StackFrame frame>       (. int expType, expCount, labelCount;
                                           bool branchNeeded = false;
                                           Label nextSwitch = new Label(!known);
                                           Label switchExit = new Label(!known);
                                           List<int> labelList = new List<int>();  .)
```

```
=  "switch" "("
     Expression<out expType>            (. if (IsArray(expType) || expType == Types.noType)
                                             SemError("invalid selector type"); .)
   ")"
   "{"
      {                                 (. if (branchNeeded) CodeGen.branch(switchExit);
                                           branchNeeded = true;
                                           nextSwitch.Here();
                                           nextSwitch = new Label(!known);
                                           CodeGen.duplicate(); .)
        CaseLabelList<out labelCount, expType, labelList>
                                        (. CodeGen.Membership(labelCount, expType);
                                           CodeGen.BranchFalse(nextSwitch); .)
        Statement<frame, switchExit, null>
        { Statement<frame, switchExit, null> }
      }
      (   "default" ":"                 (. if (branchNeeded) CodeGen.Branch(switchExit);
                                           nextSwitch.Here(); .)
        { Statement<frame, switchExit, null> }
      |                                 (. nextSwitch.Here(); .)
      )
   "}"                                  (. switchExit.Here();
                                           CodeGen.Pop(1); .) .


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

CaseLabel<. int expType, List<int> labelList .>
                                        (. ConstRec con;
                                           int factor = 1;
                                           bool signed = false; .)
=  "case"
   [ ( "+" | "-"                        (. factor = -1; .)
     )                                  (. signed = true; .)
   ] Constant<out con> ":"              (. if (!Compatible(con.type, expType)
                                                || signed && con.type != Types.intType)
                                             SemError("invalid label type");
                                           int lab = factor * con.value;
                                           if (labelList.Contains(lab))
                                             SemError("duplicated case label");
                                           else labelList.Add(lab);
                                           CodeGen.LoadConstant(lab); .) .
```

Notes

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

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

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

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

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

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

could be used as a means of providing the "fall through" semantics of some other versions of the *SwitchStatement*.

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

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

which gives rise to code like

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

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

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

and its interpretation is as follows:

```
      case PVM.dup:           // duplicate top of stack
        tos = Pop();
        Push(tos); Push(tos);
        break;
```

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

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

with a two-word opcode interpreted as follows:

```
case PVM.memb:          // membership test
  bool isMember = false;
  loop = Next();
  int test = mem[cpu.sp + loop];
  for (int m = 0; m < loop; m++) // must look at entire list - why?
    if (Pop() == test) isMember = true;
  mem[cpu.sp] = isMember ? 1 : 0;
  break;
```

As a wannabe computer language designer/lawyer (who isn't, after taking this course?) you might like to ponder whether it is a Good Idea to use the same key word "break" to denote two different kinds of breakout from a structured statement.