Computer Science 3 - 2011

Programming Language Translation

Practical 7 - Week beginning 19 May 2014 - solutions

Sources of full solutions for these problems may be found on the course web page as the file `PRAC7A.ZIP` (Java) or `PRAC7AC.ZIP` (C#).

Task 2 - Use of the debugging and other pragmas

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

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

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

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

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

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

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

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

Task 3 - How long is a piece of string?

The prac sheet asked why languages generally impose a restriction that a literal string must be contained on a single line of code. The reason is quite simple - it becomes difficult to see or track the control characters and spaces that would otherwise be buried in the string. It is easier and safer for language designers to use the escape sequence idea if they need to cater for non-graphic characters in strings and character literals.

Concatenating strings is simple. The place to do it is in the *StringConst* production which calls on a *OneString* parser to obtain the substrings (which have had their leading quotes and internal escape characters processed by the time the concatenation takes place):

```
StringConst<out String str>           (. String str2; .)
= OneString<out str>
  { [ "+" ] OneString<out str2>       (. str = str + str2; .)
  } .

OneString<out String str>
=  stringLit                          (. str = token.val;
                                         str = unescape(str.substring(1, str.length() - 1)); .) .
```

Task 4 - Things are not always what they seem

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

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

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

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

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

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

Task 5 - 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 boolean declared = true;       // true for all except sentinel entry
*   public boolean canChange= true;       // false if value cannot be changed
  public Entry   nextInScope = null;    // link to next entry in current scope
} // 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 = Kinds.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 != Kinds.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 != Kinds.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 == Kinds.Var) CodeGen.loadAddress(entry); .)
*     [  "["                            (. des.canChange = true;
                                           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 6 - Suppressing some error messages

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

```
   Designator<out DesType des>          (. String name;
                                           int indexType; .)
   =  Ident<out name>                   (. Entry entry = Table.find(name);
*                                          boolean notDeclared = !entry.declared;
*                                          if (notDeclared) {
*                                            SemError("undeclared identifier");
*                                            entry = new Entry(); // new is critical
*                                            entry.name = name;
*                                            entry.kind = Kinds.Var;
*                                            entry.type = Types.noType;
*                                            entry.offset = 0;
*                                            Table.insert(entry);
*                                          }
                                           des = new DesType(entry);
                                           if (entry.kind == Kinds.Var) CodeGen.loadAddress(entry); .)
*     [  "["                            (. if (notDeclared) entry.type++;
*                                          else 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 7 - A task to keep you on the right lines

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 former `read()` and `write()` statements. The extensions needed to the code generator and the PVM should be obvious, but it was clear from some submissions that some of you are unfamiliar with the concept of `readLine()` as a device for ignoring the rest of a line in the *data* file (not, for heaven's sake, in the grammar!). 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 8 - 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>        (. int count;
                                           Label falseLabel = new Label(!known);
                                           Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"            (. CodeGen.branchFalse(falseLabel); .)
*        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 thing instead. This can generate `BRN` instructions where none are needed. Devoid of checking, just to save space:

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

For example, source code like

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

leads to object code like

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

Tasks 9 and 10 - This has gone on long enough - time for a break, and then continue

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.

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

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

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

```
                                          StackFrame frame = new StackFrame();
                                          Table.openScope();
                                          Label DSPLabel = new Label(known);
                                          CodeGen.openStackFrame(0); .)
*     WEAK "{" { Statement<frame, null, null> }
      WEAK "}"
```

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

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

Task 11 - Some ideas are worse than a snake in the grass

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

```
   Assignment                            (. int expType;
                                            DesType des;
*                                           int count = 0;
*                                           int desCount = 1, expCount = 1;
*                                           ArrayList<Integer> typeList = new ArrayList<Integer>(); .)
   =    Designator<out des>               (. if (des.entry.kind != Kinds.Var)
                                              SemError("invalid assignment");
*                                           if (!des.canChange)
*                                             SemError("may not alter this variable");
*                                           typeList.add(des.type); .)
        (
*          { "," Designator<out des>      (. if (des.entry.kind != Kinds.Var)
*                                             SemError("invalid assignment");
*                                           if (!des.canChange)
*                                             SemError("may not alter this variable");
*                                           typeList.add(des.type);
*                                           desCount++; .)
*          }
           AssignOp
*          Expression<out expType>         (. if (count < typeList.size()
*                                                && !assignable(typeList.get(count++), expType))
*                                             SemError("incompatible types in assignment"); .)
*          { "," Expression<out expType>  (. if (count < typeList.size()
*                                                && !assignable(typeList.get(count++), expType))
*                                              SemError("incompatible types in assignment");
*                                           expCount++; .)
*          }                              (. if (expCount != desCount)
*                                             SemError("left and right counts disagree");
*                                           CodeGen.assignN(desCount); .)
        )
       WEAK ";" .
```

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

```
public static void assignN(int n) {
// Generates code to store n values currently on top-of-stack on the n addresses
// below these, finally discarding 2*n elements from the stack.
  if (n > 1) { emit(PVM.ston); emit(n); }
  else emit(PVM.sto);
}
```

Interpretation is fairly straightforward:

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

Task 12 - Your professor is quite a character

To allow for a character type involves one in a lot of straightforward alterations, as well as some more elusive ones and few, if any, submissions were even vaguely correct. in fact, most were pretty terrible, suggesting tha many students had simply not grasped the concept of type and compatibility at all. **These are important concepts in compiling, so make sure you study what follows very carefully.**

Firstly, we extend the definition of the `Types` class:

```
   class Types {
     public static final int
       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 Types
```

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

```
 *  Types.addType("char");
```

A minor change to the *Constant* production is needed to allow character literals to be regarded as of the new `charType`. Most submissions at least got this far:

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

Reading and writing single characters is easy, unless as some submissions did, you just missed this!

```
                                        DesType des; .)
   =   StringConst<out str>             (. CodeGen.writeStr(str); .)
     | Designator<out des>              (. if (des.entry.kind != Kinds.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 Types.intType:  emit(PVM.inpi); break;
       case Types.boolType: emit(PVM.inpb); break;
*      case Types.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 Types.intType:  emit(PVM.prni); break;
       case Types.boolType: emit(PVM.prnb); break;
*      case Types.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, and annoyingly perhaps, in the C family of languages it is a sort of arithmetic type when this is convenient (this is called "auto-promotion"). Explicitly, it ranks as an arithmetic type, in that expressions of the form

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

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

```
   static boolean isArith(int type) {
*    return type == Types.intType || type == Types.charType || type == Types.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 == Types.noType || typeTwo == Types.noType
            || isArray(typeOne) && typeTwo == Types.nullType
            || isArray(typeTwo) && typeOne == Types.nullType;
   }
```

The preceding discussion relates to *expression compatibility*. 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 with the aid of a further helper method, `assignable()`. I don't recall any submission making this distinction, preferring to tweak `compatible` incorrectly.

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

The `assignable()` function call now takes the place of the `compatible()` function call in the many places in *OneVar* and *AssignmentStatement* 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

A great many submissions made the mistake of thinking that a cast can only appear in the context of a simple assignment, that is, restricted to statements like

```
char  ch;
int x, y, z;

ch = (char) SomeExpression
ch = (char) x + y + z;  // understood wrongly to "mean"
                        // ch = (char) (x + y + z);  // a character assignment
```

but that is not the case. Casting applies to a component of an expression, so that the above really "means":

```
ch = ((char) x)  + y + z;    // an incorrect integer assignment again
                             // integer expressions are incompatible with
                             // character target designators
```

and as a further example it is quite legal to write

```
boolean b = 'A' < (char) (x + (int) 'B') ;
```

Of course, the C family syntax is crazy (in spite of what some of my colleagues think). It would have been infinitely better to use a notation like

```
boolean b = 'A' > char(x + int('B'));
```

but they would not have been able to do that easily (do you see why? It might be an exam question; you never know my devious mind).

To get it right requires that casting be 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 ")"`:

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

```
Primary<out int type>                (. type = Types.noType;
                                        int size;
                                        DesType des;
                                        ConstRec con; .)
=    Designator<out des>             (. type = des.type;
                                        switch (des.entry.kind) {
                                          case Kinds.Var:
                                            CodeGen.dereference();
                                            break;
                                          case Kinds.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 = Types.charType;
*                                        CodeGen.castToChar(); .)
*      | "int" ")"
*        Factor<out type>            (. if (!isArith(type))
*                                         SemError("invalid cast");
*                                        else type = Types.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:

```
AddExp<out int type>                        (. int type2;
                                               int op; .)
=  MultExp<out type>
   { AddOp<out op>
     MultExp<out type2>                     (. if (!isArith(type) || !isArith(type2)) {
                                                 SemError("arithmetic operands needed");
                                                 type = Types.noType;
                                               }
*                                              else type = Types.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);
} // CodeGen.castToChar
```

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 13 - Absolutely, yes, ja, yebo …

Add a simple `abs` function to the expression grammar (take care to get the precedence and semantic checking correct, as it is trivial to add it to the grammar. The code generator and PVM will need some simple extensions too, of course.

```
int
  i = abs(-24),
  j = abs(abs(i) * abs(-24 * i));
```

Once again, this is all easily achieved by additions to the options in the *Primary* production. Note the auto-promotion to integer type regardless. Do you suppose one should promote to `noType` instead?

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

The code generator is easily extended

```
public static void abs() {
// Generates code to leave abs(tos) on top of stack
  emit(PVM.abs);
} // CodeGen.abs
```

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

```
case PVM.abs:           // replace TOS by its absolute value
  if (mem[cpu.sp] < 0) mem[cpu.sp] = - mem[cpu.sp];
  break;
```

Task 14 - What are we doing this for?

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

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

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

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

should be regarded as more closely equivalent to

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

These arrangements can be handled by the following parsing method

```
ForStatement<StackFrame frame>        (. int expType;
                                         boolean savedCanChange;
                                         Label loopBody = new Label(!known);
                                         Label loopExit = new Label(!known);
                                         Label loopContinue = new Label(!known);
                                         String name; .)
=  "for" Ident<out name>               (. Entry var = Table.find(name);
                                         if (!var.declared)
                                           SemError("undeclared identifier");
                                         if (var.kind != Kinds.Var)
                                           SemError("illegal control variable");
                                         if (!var.canChange)
                                           SemError("may not alter this variable");
                                         if (!isArith(var.type))
                                           SemError("control variable must be of arithmetic type");
                                         CodeGen.loadAddress(var);
                                         savedCanChange = var.canChange;
                                         var.canChange = false; .)
     AssignOp
     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, loopContinue>
                                       (. loopContinue.here();
                                         CodeGen.endForLoop(up, loopBody);
                                         var.canChange = savedCanChange;
                                         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());
} // CodeGen.startForLoop

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());
} // CodeGen.endForLoop

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

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

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

Notes

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

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

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

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

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

  which generates code

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

- This solution allows for the introduction of guards againt accidental or deliberate corruption of the control variable within the *Statement* that forms the body of the loop.

- 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 .
```

- Why do you suppose the *ForStatement* has been specified to have an *Identifier* as a control variable and not a *Designator*?

Task 15 - Make the change; enjoy life; upgrade now to Parva++ (Ta-ra!)

It might not at first have been obvious, but I had fondly hoped that everyone eventually saw that this extension is handled at the initial level by clever modifications to the *AssignmentStatement* production, which has to be factorized in such a way as to avoid LL(1) conflicts. (alas, I was wrong again. Didn't we discuss this very point in lectures and in prac 21?) The code below handles this task (including the tests for compatibility and for the designation of variables rather than constants that several students omitted) by assuming the existence of a few new machine opcodes, as suggested in the textbook.

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

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

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

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

```
  case PVM.inc:           // int ++
    adr = pop();
    if (inBounds(adr)) mem[adr]++;
    break;
  case PVM.dec:           // int --
    adr = pop();
    if (inBounds(adr)) mem[adr]--;
    break;
  case PVM.incc:          // char ++
    adr = pop();
    if (inBounds(adr))
      if (mem[adr] < maxChar) mem[adr]++;
      else ps = badVal;
    break;
  case PVM.decc:          // char --
    adr = pop();
    if (inBounds(adr))
      if (mem[adr] > 0) mem[adr]--;
      else ps = badVal;
    break;
```

Task 14 - An Alternative for, for which I am grateful

Every year some students come up with alternatve solutions to my own which have distinct merit. So it was that this year Trevor Chikambura and Courtney Pitcher tried to implement the *for* statement rather more directly in terms of the suggested semantics:

```
        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:
```

Their solution was not quite complete, but here are their ideas cleaned up a little, and this will be added, with acknowledgement, to the archive I have built up of happy times spent with CSC 301.

```
// Variation based on submission by Courtney Pitcher and Trevor Chikambura, 2014

  FORStatement<StackFrame frame>            (. int expType;
                                               boolean savedCanChange;
                                               Label loopStart = new Label(!known);
                                               Label loopExit  = new Label(!known);
                                               Label loopContinue = new Label(!known);
                                               String name; .)
  =  "FOR" Ident<out name>                  (. Entry var = Table.find(name);
                                               if (!var.declared)
                                                 SemError("undeclared identifier");
                                               if (var.kind != Kinds.Var)
                                                 SemError("illegal control variable");
                                               if (!var.canChange)
                                                 SemError("may not alter this variable");
                                               if (!IsArith(var.type))
                                                 SemError("control variable must be of arithmetic type");
```

```
                                        savedCanChange = var.canChange;
                                        var.canChange = false; .)
  AssignOp 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.checkAnyIterationsNeeded(up);
                                        CodeGen.branchFalse(loopExit);
                                        CodeGen.loadAddress(var);
                                        CodeGen.assign2(var.type);
                                        loopStart.here(); .)
  Statement<frame, loopExit, loopContinue>
                                        (. loopContinue.Here();
                                        CodeGen.loadAddress(var);
                                        CodeGen.dereference();
                                        CodeGen.checkMoreIterations();
                                        CodeGen.branchFalse(loopExit);
                                        CodeGen.loadAddress(var);
                                        CodeGen.incOrDec(up, var.type);
                                        CodeGen.branch(loopStart);
                                        loopExit.here();
                                        var.canChange = savedCanChange;
                                        CodeGen.pop(2); .) .
```

This solution relies on the presence of four new opcodes:

```
      case PVM.sto2:          // store sos on address found on top of stack (do not pop the value)
        adr = Pop();
        if (InBounds(adr)) mem[adr] = mem[cpu.sp + 1];
        break;

      case PVM.cfu:           // check whether a "to" for loop will execute at least once
        Push(mem[cpu.sp + 1] <= mem[cpu.sp] ? 1 : 0);
        break;

      case PVM.cfd:           // check whether a "downto" for loop will execute at least once
        Push(mem[cpu.sp + 1] >= mem[cpu.sp] ? 1 : 0);
        break;

      case PVM.ckfin:         // check whther a for loop has completed all iterations
        Push(Pop() != mem[cpu.sp] ? 1 : 0);
        break;
```

and by now you should be able to predict the simple modifications needed to the code generator!