Computer Science 3 - 2009

Programming Language Translation

Practical for Weeks 25 - 26, beginning 19 October 2009 - solutions

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

Task 2 - Better use of the debugging pragma

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

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

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

```
PRAGMAS
  WarnOn      = "$W+" .         (. warnings = true; .)
  WarnOff     = "$W-" .         (. warnings = false; .)
  CodeOn      = "$C+" .         (. listCode = true; .)
  CodeOff     = "$C-" .         (. listCode = false; .)
  OptimizeOn  = "$O+" .         (. optimize = true; .)
  OptimizeOff = "$O-" .         (. optimize = false; .)
```

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

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

Task 5 - Detecting other meaningless forms of code

A joint answer seems in order. Spotting the empty statement in the form of a stray semicolon is only part of the solution. Detecting blocks that really have no effect might be handled in several ways. The suggestion below counts the executable statements in a *Block*. This means that the *Statement* parser has to be attributed so as to return this count, and this has a knock-on effect in various other productions as well. Since we might have all sorts of nonsense like

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

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

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

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

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

Task 6 - Some simple statement extensions

The extensions to the *WriteStatement* and *HaltStatement* are very simple. It is useful to allow a `WriteLine()` statement - that is, one with no *WriteElements*.

```
    HaltStatement                         (. String str; .)
    =  "halt"
**      [ "(" StringConst<out str>        (. CodeGen.writeString(str); .)
**        ")" ]                           (. CodeGen.leaveProgram(); .)
        WEAK ";" .

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

Task 7 - You had better do this one or else….

Adding an *else* option to the *IfStatement* is easy once you see the trick! Note that the "no else part" option in the grammar is associated with an action, even in the absence of any terminals or non-terminals. This is a very useful trick to remember.

```
IfStatement<StackFrame frame>          (. int count;
                                          Label falseLabel = new Label(!known); .)
=  "if" "(" Condition ")"               (. CodeGen.branchFalse(falseLabel); .)
   Statement<out count, frame>         (. if (count == 0 && warnings)
                                            Warning("empty statement part"); .)
**    (   "else"                       (. Label outLabel = new Label(!known);
**                                        CodeGen.branch(outLabel);
**                                        falseLabel.here(); .)
**        Statement<out count, frame>  (. if (count == 0 && warnings)
**                                            Warning("empty statement part");
**                                        outLabel.here(); .)
**      | /* no else part */           (. falseLabel.here(); .)
      ) .
```

Task 8 - Something to do - while you wait for a tutor

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

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

Task 9 - This has gone on long enough - time for a break

The syntax of the *BreakStatement* (and of the similar *ContinueStatement*) is, of course, trivial. The catch is that one has to allow these statements only in the context of loops. To find a context-free grammar with this restriction is not worth the effort. As with nested comments in languages that allow them, it is easier just to have a (global) counter that is incremented and decremented as parsing of loops starts and finishes. Although you were asked only to deal with *break* it is easy enough to illustrate *continue* as well for your enlightenment!

For some time I have advocated the following approach:

Loops must be handled in a way that allows them to be nested, with all the *breaks* and *continues* in each loop directed at the correct place for that loop - and many of these involve forward references. As it happens, the `Label` class we already use allows for this to be handled neatly, and we can get away with using two global labels (one for each kind of statement). However, we need a little local stack to be introduced in each loop parsing production, so that these global labels can be kept up to date. Once you have seen the solution it probably looks almost obvious!

The extra static fields in the parser (declared at the top of the ATG file are;

```
static int loopLevel = 0;                  // = 0 outside of loops, > 0 inside loops
static Label
  loopExit = new Label(!known),            // current target for "break" statements
  loopContinue = new Label(!known);        // current target for "continue" statements
```

and the productions for the *BreakStatement* and *ContinueStatement* follow as:

```
BreakStatement
=  "break"                             (. if (loopLevel == 0) SemError("break is not within a loop");
                                          else CodeGen.branch(loopExit); .)
   WEAK ";" .
```

```
    ContinueStatement
    =  "continue"                        (. if (loopLevel == 0) SemError("continue is not within a loop");
                                            else CodeGen.branch(loopContinue); .)
       WEAK ";" .
```

The *WhileStatement* and *DoWhileStatement* productions now have quite a lot of extra actions:

```
    WhileStatement<StackFrame frame>     (. int count;
**                                          loopLevel++;
**                                          Label oldContinue = loopContinue;
**                                          Label oldExit = loopExit;
**                                          loopExit = new Label(!known);
**                                          loopContinue = new Label(known); .)
    =  "while" "(" Condition ")"         (. CodeGen.branchFalse(loopExit); .)
       Statement<out count, frame>       (. if (count == 0 && warnings)
                                              Warning("empty statement part");
                                          CodeGen.branch(loopContinue);
**                                        loopExit.here();
**                                        loopExit = oldExit;
**                                        loopContinue = oldContinue;
**                                        loopLevel--; .)
    .

    DoWhileStatement<StackFrame frame>   (. int count;
**                                          loopLevel++;
**                                          Label oldContinue = loopContinue;
**                                          Label oldExit = loopExit;
**                                          loopContinue = new Label(!known);
**                                          Label startLoop = new Label(known);
**                                          loopExit = new Label(!known); .)
    =  "do"
         Statement<out count, frame>     (. if (count == 0 && warnings)
                                              Warning("empty statement part"); .)
       WEAK "while"                      (. loopContinue.here(); .)
       "(" Condition ")" WEAK ";"        (. CodeGen.negateBoolean();
                                          CodeGen.branchFalse(startLoop);
**                                        loopExit.here();
**                                        loopExit = oldExit;
**                                        loopContinue = oldContinue;
**                                        loopLevel--; .)
    .
```

Another solution, which some might think of, dispenses with the counter by initializing `loopExit` to `null`:

```
     static Label loopExit = null;                // current target for "break" statements
```

and the production for the *BreakStatement* follows as

```
    BreakStatement
**  =  "break"                           (. if (loopExit == null) SemError("break is not within a loop");
**                                          else CodeGen.branch(loopExit); .)
**     WEAK ";" .
```

and the production for the *DoWhileStatement* simplifies to:

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

However, prompted by various attempts made by members of this year's class, I have come to realize that an alternative approach, based on passing labels as arguments to various subparsers is actually a better and more elegant solution. (As I have said before, I learn something every year from students, and it is always very exciting to me when that happens, so well done!)

In this approach we change the parser for `Statement` and for `Block` as follows:

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

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

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

```
    IfStatement<StackFrame frame, Label breakLabel, Label continueLabel>
                                        (. int count;
                                           Label falseLabel = new Label(!known); .)
    =  "if" "(" Condition ")"           (. CodeGen.branchFalse(falseLabel); .)
**     Statement<out count, frame, breakLabel, continueLabel>
                                        (. if (count == 0 && warnings)
                                             Warning("empty statement part"); .)
       (   "else"                       (. Label outLabel = new Label(!known);
                                           CodeGen.branch(outLabel);
                                           falseLabel.here(); .)
**        Statement<out count, frame, breakLabel, continueLabel>
                                        (. if (count == 0 && warnings)
                                             Warning("empty statement part");
                                           outLabel.here(); .)
         | /* no else part */           (. falseLabel.here(); .)
       ) .

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

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

```
**  BreakStatement<Label breakLabel>
    =  "break"                          (. if (breakLabel == null) SemError("break is not within a loop");
**                                         else CodeGen.branch(breakLabel); .)
       WEAK ";" .

**  ContinueStatement<Label continueLabel>
    =  "continue"                       (. if (continueLabel == null) SemError("continue is not within a loop");
**                                         else CodeGen.branch(continueLabel); .)
       WEAK ";" .
```

Task 10 - Your professor is quite a character

To allow for a character type involves one in a lot of straightforward alteratons, as well as some more elusive ones. Firstly, we extend the definition of a symbol table entry:

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

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

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

```
         static String[] typeName = {
           "none", "none[]", "null", "null[]", "int ", "int[] ",
******     "bool", "bool[]", "char", "char[]", "void", "void[]" };
```

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

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

Reading and writing single characters is easy:

```
    ReadElement                         (. String str;
                                           DesType des; .)
    =   StringConst<out str>            (. CodeGen.writeString(str); .)
      | Designator<out des>             (. if (des.entry.kind != Entry.Var)
                                             SemError("wrong kind of identifier");
                                           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.writeString(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 have similar changes:

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

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

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

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

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

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

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

However, assignment compatibility is more restricted

integer = integer
integer = character
character = character

is allowed, but

character = integer

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

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

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

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

character = (char) integer

is allowed, and for completeness, so are

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

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

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

```
Primary<out int type>                (. type = Entry.noType;
                                        int size;
                                        DesType des;
                                        ConstRec con; .)
=    Designator<out des>             (. type = des.type;
                                        switch (des.entry.kind) {
                                          case Entry.Var:
                                            CodeGen.dereference();
                                            break;
                                          case Entry.Con:
                                            CodeGen.loadConstant(des.entry.value);
                                            break;
                                          default:
                                            SemError("wrong kind of identifier");
                                            break;
                                        } .)
   | Constant<out con>               (. type = con.type;
                                        CodeGen.loadConstant(con.value); .)
   | "new" BasicType<out type>       (. type++; .)
     "[" Expression<out size>        (. if (!isArith(size))
                                          SemError("array size must be integer");
                                        CodeGen.allocate(); .)
     "]"
   | "("
**   (   "char" ")"
**       Factor<out type>            (. if (!isArith(type))
**                                        SemError("invalid cast");
**                                      else type = Entry.charType;
**                                      CodeGen.castToChar(); .)
**     | "int" ")"
**       Factor<out type>            (. if (!isArith(type))
**                                        SemError("invalid cast");
**                                      else type = Entry.intType; .)
**     | Expression<out type> ")"
**   )
.
```

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

Various of the other productions need modification. The presence of an arithmetic operator correctly placed between character or integer operands must result in the sub-expression so formed being of integer type (and never of character type). So for example:

```
MultExp<out int type>                (. int type2;
                                        int op; .)
=  Factor<out type>
   { MulOp<out op>
     Factor<out type2>               (. if (!isArith(type) || !isArith(type2)) {
                                          SemError("arithmetic operands needed");
                                          type = Entry.noType;
                                        }
**                                      else type = Entry.intType;
                                        CodeGen.binaryOp(op); .)
   } .
```

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

The code generation method we need is as follows:

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

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

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

The interpreter has another opcode for checked storage of characters, but if the `i2c` opcodes are inserted correctly it does not appear that we really need to use this:

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

Task 11 - Make the change; enjoy life; upgrade now to Parva++ (Ta-ra!)

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

```
Assignment                          (. int expType;
                                       DesType des; .)
=  Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                         SemError("invalid assignment"); .)
**    (  AssignOp
**       Expression<out expType>    (. if (!assignable(des.type, expType))
**                                     SemError("incompatible types in assignment");
**                                   CodeGen.assign(des.type); .)
**      | "++"                      (. if (!isArith(des.type))
**                                     SemError("arithmetic type needed");
**                                   CodeGen.increment(des.type); .)
**      | "--"                      (. if (!isArith(des.type))
**                                     SemError("arithmetic type needed");
**                                   CodeGen.decrement(des.type); .)
**    )
      WEAK ";" .
```

The extra code generation routines are straightforward:

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

public static void decrement(int type) {
// Generates code to decrement the value found at the address currently
// stored at the top of the stack.
// If necessary, apply character range check
  if (type == Entry.charType) emit(PVM.decc); else emit(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 12 - All assignments have equals, some have more equals than others

These extensions are handled by further modifications to the *Assignment* production. The code below shows this (including the tests for compatibility that some students omitted) by assuming the existence of the `DUP` opcode, as suggested in the textbook.

For the boolean assignment operators `&=` and `|=` we have to make sure that we generate "short circuit" operations. This adds a bit to the apparent complexity of the production, as you can see below:

```
      Assignment                          (. int expType;
                                             DesType des;
                                             int op;
                                             Label shortcircuit = new Label(!known);.)
      =  Designator<out des>              (. if (des.entry.kind != Entry.Var)
                                               SemError("invalid assignment"); .)
**      (  AssignOp<out op>               (. if (op != CodeGen.nop) {
**                                           CodeGen.duplicate();
**                                           CodeGen.dereference();
**                                           switch(op) {
**                                             case CodeGen.add:
**                                             case CodeGen.sub:
**                                             case CodeGen.mul:
**                                             case CodeGen.div:
**                                             case CodeGen.rem:
**                                               if (!isArith(des.type))
**                                                 SemError("arithmetic destination required");
**                                               break;
**                                             case CodeGen.and:
**                                             case CodeGen.or:
**                                               if (!isBool(des.type))
**                                                 SemError("boolean destination required");
**                                               CodeGen.booleanOp(shortcircuit, op);
**                                               break;
**                                           }
**                                         } .)
**        Expression<out expType>         (. if (!assignable(des.type, expType))
**                                             SemError("incompatible types in assignment");
**                                           switch (op) {
**                                             case CodeGen.nop:
**                                               break;
**                                             case CodeGen.and:
**                                             case CodeGen.or:
**                                               shortcircuit.here();
**                                               break;
**                                             case CodeGen.add:
**                                             case CodeGen.sub:
**                                             case CodeGen.mul:
**                                             case CodeGen.div:
**                                             case CodeGen.rem:
**                                               CodeGen.binaryOp(op);
**                                               break;
**                                           }
**                                           CodeGen.assign(des.type); .)
```

```
      | "++"                          (. if (!isArith(des.type))
                                           SemError("arithmetic destination required");
                                        CodeGen.increment(des.type); .)
      | "--"                          (. if (!isArith(des.type))
                                           SemError("arithmetic destination required");
                                        CodeGen.decrement(des.type); .)
    )
    WEAK ";" .
```

The *AssignOp* production is rather more complex than before. It returns the associated binary operation that will be needed in the compound assignment operations.

```
AssignOp<out int op>                (. op = CodeGen.nop; .)
=    "="
    | ":="                          (. SemError("= intended?"); .)
    | "+="                          (. op = CodeGen.add; .)
    | "-="                          (. op = CodeGen.sub; .)
    | "*="                          (. op = CodeGen.mul; .)
    | "/="                          (. op = CodeGen.div; .)
    | "%="                          (. op = CodeGen.rem; .)
    | "&="                          (. op = CodeGen.and; .)
    | "|="                          (. op = CodeGen.or;  .)
.
```

The extra code generation routine needed is straightforward:

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

As usual, the extra opcodes in the PVM make all this easy to achieve at run time. Many submission forgot 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.dup:           // duplicate top of stack
  cpu.sp--;
  if (inBounds(cpu.sp)) mem[cpu.sp] = mem[cpu.sp + 1];
  break;
```

Task 13 - Generating slightly better code

The changes to the code generating routines to produce the special one-word opcodes like `LDA_0` and `LDC_3` are very simple:

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

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

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

Task 13a - Generating much better code

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

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

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

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

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

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

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

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

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

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

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

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

```
      Assignment                           (. int expType;
                                              DesType des;
                                              int op;
                                              Label shortcircuit = new Label(!known);.)
      =  Designator<out des>               (. if (des.entry.kind != Entry.Var)
                                                SemError("invalid assignment"); .)
         (  AssignOp<out op>               (. if (op != CodeGen.nop) {
**                                              if (des.isSimple)
**                                                CodeGen.loadValue(des.entry);
**                                              else {
**                                                CodeGen.duplicate();
**                                                CodeGen.dereference();
**                                              }
                                                switch(op) {
                                                  case CodeGen.add:
                                                  case CodeGen.sub:
                                                  case CodeGen.mul:
                                                  case CodeGen.div:
                                                  case CodeGen.rem:
                                                    if (!isArith(des.type))
                                                      SemError("arithmetic destination required");
                                                    break;
                                                  case CodeGen.and:
                                                  case CodeGen.or:
                                                    if (!isBool(des.type))
                                                      SemError("boolean destination required");
                                                    CodeGen.booleanOp(shortcircuit, op);
                                                    break;
                                                }
                                              } .)
            Expression<out expType>        (. if (!assignable(des.type, expType))
                                                SemError("incompatible types in assignment");
                                              switch (op) {
                                                case CodeGen.nop:
                                                  break;
                                                case CodeGen.and:
                                                case CodeGen.or:
                                                  shortcircuit.here();
                                                  break;
                                                case CodeGen.add:
                                                case CodeGen.sub:
                                                case CodeGen.mul:
                                                case CodeGen.div:
                                                case CodeGen.rem:
                                                  CodeGen.binaryOp(op);
                                                  break;
                                              }
**                                            if (des.isSimple) CodeGen.storeValue(des.entry);
**                                            else CodeGen.assign(des.type); .)
**          | "++"                         (. if (des.isSimple) CodeGen.loadAddress(des.entry);
                                              if (!isArith(des.type))
                                                SemError("arithmetic type needed");
                                              CodeGen.increment(des.type); .)
**          | "--"                         (. if (des.isSimple) CodeGen.loadAddress(des.entry);
                                              if (!isArith(des.type))
                                                SemError("arithmetic type needed");
                                              CodeGen.decrement(des.type); .)
         )
         WEAK ";" .
```

The code generating routines needed are

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

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

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