Computer Science 3 - 2015

Programming Language Translation

Practical 7 - Week beginning 19 May 2015 - solutions

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

Task 2 - Use of the debugging and other pragmas

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

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

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

```
   PRAGMAS
     DebugOn     = "$D+" .              (. debug     = true; .)
     DebugOff    = "$D-" .              (. debug     = false; .)
*    ShortOn     = "$S+" .              (. shortCirc = true; .)
*    ShortOff    = "$S-" .              (. shortCirc = false; .)
*    OptCodeOn   = "$O+" .              (. optimize  = true; .)
*    OptCodeOff  = "$O-" .              (. optimize  = 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].ToLower().Equals("-l")) mergeErrors = true;
        else if (args[i].ToLower().Equals("-d")) Parser.debug = true;
*       else if (args[i].ToLower().Equals("-w")) Parser.warnings = false;
*       else if (args[i].ToLower().Equals("-c")) Parser.listCode = true;
*       else if (args[i].ToLower().Equals("-o")) Parser.optimize = true;
*       else if (args[i].ToLower().Equals("-s")) Parser.shortCirc = false;
        else if (args[i].ToLower().Equals("-n")) execution = false;
        else if (args[i].ToLower().Equals("-g")) immediate = true;
        else inputName = args[i];
      }
      if (inputName == null) {
        Console.WriteLine("No input file specified");
        Console.WriteLine("Usage: Parva [-l] [-d] [-w] [-c] [-o] [-s] [-n] [-g] source.pav");
        Console.WriteLine("-l directs source listing to listing.txt");
        Console.WriteLine("-d turns on debug mode");
*       Console.WriteLine("-w suppresses warnings");
*       Console.WriteLine("-c lists object code (.cod file)");
*       Console.WriteLine("-o optimised code");
*       Console.WriteLine("-s suppresses short circuit evaluation");
        Console.WriteLine("-n no execution after compilation");
        Console.WriteLine("-g execute immediately after compilation (StdIn/StdOut)");
        System.Environment.Exit(1);
      }
```

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

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

Task 3 - 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:

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

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

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 would mean that the *Statement* parser had 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 4 - You had better do this one or else….

The problem, firstly, asked for the addition of an *else* option to the *IfStatement*. Adding an *else* option to the *IfStatement* efficiently is easy once you see the trick. Note the use of the "no else part" option associated with an action, even in the absence of any terminals or non-terminals. This is a very useful technique to remember.

```
IfStatement<StackFrame frame>        (. Label falseLabel = new Label(!known);
                                        Label outLabel = new Label(!known); .)
=  "if" "(" Condition ")"            (. CodeGen.BranchFalse(falseLabel); .)
*       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(); .) .
```

Using this strategy, source code like

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

would lead to object code like

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

Handling the *elsif* clauses uses the same sort of idea. Note that. after defining `falseLabel.Here()`, the label is "re-used" by assigning it another instance of an "unknown" label. If you don't do this you will get all sorts of

bad code or funny messages from the label handler!

```
   IfStatement<StackFrame frame)
                                       (. Label falseLabel = new Label(!known);
                                          Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"           (. CodeGen.BranchFalse(falseLabel); .)
         Statement<frame)
*     {                                (. CodeGen.Branch(outLabel);
*                                         falseLabel.Here();
*                                         falseLabel = new Label(!known); .)
*       "elsif" "(" Condition ")"      (. CodeGen.BranchFalse(falseLabel); .)
*          Statement<frame>
*     }
      (  "else"                        (. CodeGen.Branch(outLabel);
                                          falseLabel.Here(); .)
          Statement<frame
        | /* no else part */           (. falseLabel.Here(); .)
      )                                (. outLabel.Here(); .) .
```

Task 5 Something to do - while you wait for inspiration

This was discussed in class, and I am surprised that some people had trouble with it. We need a single label, and a single conditional branch that goes to the start of the loop body. The only trick is that we don't have a "branch on true" opcode - but all we have to do is to generate a "negate boolean" operation that will be applied to the computed value of the *Condition* at run time before the conditional branch takes effect:

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

Task 6 - Repeat discussions with your team mates until you all get the idea

This one is almost exactly the same, save that one has no need of the Boolean negation operation. They don't get much easier than this.

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

Task 7 - This has gone on long enough - time for a break (and then continue)

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

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

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

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

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

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

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

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

```
   IfStatement<StackFrame frame, Label breakLabel, Label continueLabel>
                                       (. Label falseLabel = new Label(!known);
                                          Label outLabel = new Label(!known); .)
   =  "if" "(" Condition ")"           (. CodeGen.BranchFalse(falseLabel); .)
*       Statement<frame, breakLabel, continueLabel>
      {                                (. CodeGen.Branch(outLabel);
                                          falseLabel.Here();
                                          falseLabel = new Label(!known); .)
        "elsif" "(" Condition ")"      (. CodeGen.BranchFalse(falseLabel); .)
*          Statement<frame, breakLabel, continueLabel>
      }
      (  "else"                        (. CodeGen.Branch(outLabel);
                                          falseLabel.Here(); .)
*        Statement<frame, breakLabel, continueLabel>
       | /* no else part */            (. falseLabel.Here(); .)
      )                                (. outLabel.Here(); .) .

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

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

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

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

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

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

Task 8 - Your professor is quite a character

The major part of this extension is 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 != integer
character > character	integer == character
character + integer	integer + character
character > integer	integer > character

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

```
   static bool IsArith(int type) {
*    return type == Types.intType || type == Types.charType || type == Types.noType;
   }

   static bool 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()`.

```
*  static bool 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 several places in

OneVar and *AssignmentStatement* where, previously, calls to `Compatible()` appeared.

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

```
(char) integer     // returns character with internal value of integer
(int)  character   // returns integer with internal value for character
```

is allowed, and for completeness, so are the following redundant operations

```
(int)  integer
(char) character
```

A great many submissions make the mistake of thinking that a cast can only appear in the context of a simple assignment, that is, must be restricted to being found in 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 only to a **component** of an *Expression*, so that the above "means":

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

and as a further example, stressing the importance of thinking of casting as belonging within expressions, it is quite legal to write

```
bool 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 have been allowed to use a notation like

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

but they would not have been able to define that form 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 like `"(" "int" ")"`, `"(" "char" ")"` and `"(" `*Expression*` ")"`:

Casting operations are accompanied by a type *check* (you cannot cast "anything", only arithmetic values) and a type *conversion* (you fool the compiler into it is dealing with a different type for this component of an expression). However, since the PVM sores both characters and integers as 32 bit values. the `(har)` cast should generate code for checking that the *run-time* integer value to be "converted" lies within the range 0 .. 255 (the 8-bit ASCII character set).

```
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 C family 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 either an integer or a 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 (char) cast is in range
  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 might never really need `stoc`. Think about this in a quiet moment.

```
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 9 - Let's dive in where C programmers fear to go

The problem suggested that Parva might be extended to provide a relational operator for determining whether the value of an expression matched one of a set of values in a list.

We add the *in* operator into the hierarchy at the same level as the other equality operators, although of course the syntax is somewhat different:

```
EqlExp<out int type>                  (. int type2;
                                         int op;
                                         int count; .)
=  RelExp<out type>
   {   EqualOp<out op>
       RelExp<out type2>              (. if (!Compatible(type, type2))
                                           SemError("incomparable operand types");
                                         CodeGen.Comparison(op, type);
                                         type = Types.boolType; .)
*      | "in" ExpList<out count, type>  (. CodeGen.Membership(count, type);
*                                        type = Types.boolType; .)
   } .
```

Parsing the *ExpList* must include the check that all the members of the list are of a type compatible with the value of the *Expression* that is to be tested for membership. *ExpList* also returns the number of elements in the list, and generates code to push each of the values of the expressions onto the stack.

```
*  ExpList<out int count, int type>       (. int type2;
*                                            count = 1; .)
*  =  "("
*       Expression<out type2>             (. if (!Assignable(type, type2))
*                                              SemError("incompatible types"); .)
*       { "," Expression<out type2>       (. if (!Assignable(type, type2))
*                                              SemError("incompatible types");
*                                            count++; .)
*         }
*     ")" .
```

Note that the compatibility tests are done after each *Expression* has been parsed, and not left until the end. A special opcode is used to test for membership. Why do we need to insist on assignment compatibility? Code generation is easy, noting that an *ExpList* with only one element can be handled by a test for equality, using the existing opcode.

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

If the *ExpList* has more than one member, interpretation requires that the value of each of the expressions is popped off the stack and checked to see whether it matches the value being tested for membership. Finally, after the last value is popped, a pseudo-Boolean value is left on top of the stack:

```
case PVM.memb:          // membership test
  bool isMember = false;
  loop = Next();                      // the number to test
  int test = mem[cpu.sp + loop];      // the value being sought
  for (int m = 0; m < loop; m++) if (Pop() == test) isMember = true;
  mem[cpu.sp] = isMember ? 1 : 0;
  break;
```

Task 10 - What are we doing this for?

As an alternative to the Pascal-style *ForStatement*, the problem suggested implementing one modelled on a syntax suggested by Python:

```
ForStatement = "for" Ident "in" "(" Expression { "," Expression } ")" Statement .
```

This is fairly straightforward, as we already have the mechanisms for dealing with lists of expressions. We do,

however, need another pair of code generating routines and additional opcodes in the PVM:

```
*  ForStatement<StackFrame frame>          (. int expCount;
*                                             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");
*                                             CodeGen.LoadAddress(var);  .)
*        "in" ExpList<out expCount, var.type>
*                                          (. CodeGen.LoadConstant(expCount);
*                                             Label loopBody = new Label(known);
*                                             CodeGen.NextControl(expCount); .)
*        Statement<frame, loopExit, loopContinue>
*                                          (. loopContinue.Here();
*                                             CodeGen.TestFor(loopBody);
*                                             loopExit.Here();
*                                             CodeGen.ExitFor(expCount); .) .
```

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 in (35, 64, -1, 235) write(i);   // loop to be executed 4 times
```

...	4	235	-1	64	35	adr i	....

The 4 on the top of the stack will be used as a counter and decremented on each pass of the loop. The values below that will be assigned, one by one, to the designator whose address is lurking at the bottom of this segment.

The extra routines needed in the code generator are

```
public static void NextControl(int count) {
// Generates code to set the for loop control on each pass
  Emit(PVM.sfl); Emit(count);
}

public static void TestFor(Label startLoop) {
// Generates code to decrement the count after each pass through loop
// and test to see whether another iteration is needed
  Emit(PVM.tfl); Emit(startLoop.Address());
}

public static void ExitFor(int count) {
// Generates code to discard the expression list used in the for loop
  Emit(PVM.efl); Emit(count);
}
```

The interpretation of the new opcodes is as follows:

```
case PVM.sfl:               // set up the for loop control variable value
  mem[mem[cpu.sp + Next() + 1]] = mem[cpu.sp + mem[cpu.sp]];
  break;

case PVM.tfl:               // test for loop continuation
  target = Next();
  mem[cpu.sp]--;
  if (mem[cpu.sp] > 0) cpu.pc = target;
  break;

case PVM.efl:               // end for loop - clean up stack
  cpu.sp += Next() + 2;
  break;
```

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

```
for i in (one, two, three) statement
```

which generates code

```
            LDA     i
            one
            two
            three
            LDC     3
        L1  SFL     3
            statement
        L2  TFL     L1
        L3  EFL     3
```

And, of course, the solution just presented already takes into account *break* and *continue* handling!

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

It might not at first have been obvious, but I fondly hope that everyone will realize that this extension is handled at the initial level by clever modifications to the *AssignmentStatement* production, which has to be factorized in such a way as to avoid LL(1) conflicts. The code below handles this task (including the tests for assignment compatibility and for the designation of variables rather than constants that several students omitted) by assuming the existence of a few new machine opcodes, as suggested in the textbook.

```
AssignmentOrCall                        (. int expType;
                                           DesType des;
                                           bool inc = true; .)
=  Designator<out des>
     (
       "("                              (. if (des.entry.kind == Kinds.Fun
                                              && des.type == Types.voidType)
                                            CodeGen.FrameHeader();
                                           else SemError("invalid assignment or call"); .)
          Arguments<des>                (. if (des.entry.kind == Kinds.Fun)
                                            CodeGen.Call(des.entry.entryPoint); .)
        ")"

      |                                 (. if (des.entry.kind != Kinds.Var)
                                            SemError("cannot assign to "
                                                     + Kinds.kindNames[des.entry.kind]); .)
         (   AssignOp
             Expression<out expType>    (. if (!Assignable(des.type, expType))
                                           SemError("incompatible types in assignment");
                                           CodeGen.Assign(des.type); .)
           | ( "++" | "--"              (. inc = false; .)
             )                          (. if (!IsArith(des.type))
                                           SemError("arithmetic type needed");
                                           CodeGen.IncOrDec(inc, des.type); .)
         )
      )
    |  ( "++" | "--"                    (. inc = false; .)
       ) Designator<out des>            (. if (des.entry.kind != Kinds.Var)
                                            SemError("variable designator required");
                                           if (!IsArith(des.type))
                                            SemError("arithmetic type needed");
                                           CodeGen.IncOrDec(inc, des.type); .)
   ) WEAK ";" .
```

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

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

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

```
      case PVM.inc:           // integer ++
        adr = Pop();
        if (InBounds(adr)) mem[adr]++;
        break;

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

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

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

Task 12 - If you survive this I'll pass you a reference so that employers will know your value

This is remarkably easy to do. We need an extra field in the `Entry` class which will normally have the value of *false*, but will be *true* for formal parameters designated with the *ref* keyword. This modified `Entry` class has another field `canChange` whose purpose will be made clear later in the discussion of *for* loop threatening.

```
      class Entry {
      // All fields initialized, but are modified after construction (by semantic analyser)
        public int     kind        = Kinds.Var;
        public string  name        = "";
        public int     type        = Types.noType;
        public int     value       = 0;       // constants
        public int     offset      = 0;       // variables
        public bool    declared    = true;    // true for all except sentinel entry
        public Entry   nextInScope = null;    // link to next entry in current scope
        public int     nParams     = 0;       // functions
*       public bool    byRef       = false;   // true only for parameters passed by reference
        public Label   entryPoint  = new Label(false);
        public Entry   firstParam  = null;
*       public bool    canChange   = true;    // true except for extent of "for" controls
      } // end Entry
```

The `byRef` field is set true within the *OneParam* production:

```
       OneParam<out Entry param, Entry func>
       =                                    (. param = new Entry();
                                               param.kind = Kinds.Var;
                                               param.offset = CodeGen.headerSize + func.nParams;
                                               func.nParams++; .)
*        [ "ref"                            (. param.byRef = true; .) ]
         Type<out param.type>               (. if (param.type == Types.voidType)
                                                 SemError("parameters may not be of void type"); .)
         Ident<out param.name>              (. Table.Insert(param); .) .
```

Within the *OneArg* production a choice is made between pushing the address (of a reference argument) or the value of an expression (for a value argument). Note the careful check that the correct passing mechanism is being selected, as well as the assignment compatibility checks between formal and actual parameters. Finally, your attention is drawn to the lookout for *null* entries in the parameter list.

```
       OneArg<Entry fp>                     (. int argType;
                                               DesType des; .)
       =  (  Expression<out argType>        (. if (fp != null) {
                                                  if (!Assignable(fp.type, argType))
                                                   SemError("argument type mismatch");
*                                               if (fp.byRef)
*                                                 SemError("this argument must be passed by reference");
                                               } .)
```

```
*         "ref" Designator<out des>        (. if (fp != null) {
                                                if (!Assignable(fp.type, des.type))
                                                  SemError("argument type mismatch");
*                                               if (!fp.byRef)
*                                                 SemError("this argument must be passed by value");
                                              } .)
        ) .
```

If a variable has been passed "by reference" (that is, where the actual argument is its address rather than its value) then when the *Designator* parser encounters the formal parameter within the function body, an extra *dereference* opcode must be generated to be able to follow the address found as the argument to find the address of the actual variable being referenced:

```
    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);
*                                            if (entry.byRef)
*                                              CodeGen.Dereference(); .)
       [     "["                          (. if (IsArray(des.type)) des.type--;
                                             else SemError("unexpected subscript");
                                             if (des.entry.kind != Kinds.Var)
                                               SemError("unexpected subscript");
                                             CodeGen.Dereference(); .)
                Expression<out indexType>  (. if (!IsArith(indexType))
                                               SemError("invalid subscript type");
                                             CodeGen.Index(); .)

             "]"
       ] .
```

Task 13 - Time to resolve an awkward LL(1) conflict

Consider the productions which have been written

```
AssignmentOrCall = Designator ( "(" Arguments ")"  |  AssignOp Expression ) ";"  .
Designator       = identifier [ "[" Expression "]" ] .
```

That is too "loose". Syntactically this would accept something like `function[6](a,b,c)` as some sort of function call. The compiler you were given handled this problem "semantically" (make sure you understand how and why).

It would be syntactically far preferable to write productions like

```
AssignmentOrCall = ( identifier "(" Arguments ")"   |   Designator AssignOp Expression  ) ";"  .
Designator       = identifier [ "[" Expression "]" ] .
```

but then we have an LL(1) conflict because both options ultimately start with an identifier.

Coco/R has a clever way of being able to use semantic information in so-called "resolvers". This is discussed in the book, but not in much detail. Fortunately in simple cases it is quite easy to apply. Here is the basic idea: Consider a conflict like

```
LL1Problem =  One  |  Two .
```

where *FIRST(One)* and *FIRST(Two)* have a terminal in common. If we can find some semantic condition *ChooseOne* (which yields *true* for the elements of *FIRST(One)*, but *false* for the elements of *FIRST(Two)*), which would break the deadlock in favour of *One*, we code this into Cocol as follows - notice the uppercase IF, a Coco/R keyword you have not seen before.

```
LL1Problem =  IF (ChooseOne) One  |  Two .
```

In this case we can easily find such a condition. Both alternatives in *AssignmentOrCall* start with an identifier. Both identifiers should be in the symbol table; one should correspond to a function name, one to a variable name.

So here is how we resolve the issue

```
    AssignmentOrCall                          (. int expType;
                                                 DesType des;
                                                 string name;
                                                 bool inc = true; .)
*   =  (  (  IF (IsCall(out des))
*            Ident<out name>
*            "("                              (. CodeGen.FrameHeader(); .)
*               Arguments<des>                (. CodeGen.Call(des.entry.entryPoint); .)
*            ")"
*         |  Designator<out des>               (. if (des.entry.kind != Kinds.Var)
*                                                   SemError("cannot assign to "
                                                            + Kinds.kindNames[des.entry.kind]);
                                                 if (!des.canChange)
                                                   SemError("may not alter this variable");
                                                 .)
               (   AssignOp
                   Expression<out expType>       (. if (!Assignable(des.type, expType))
                                                    SemError("incompatible types in assignment");
                                                    CodeGen.Assign(des.type); .)
                 | ( "++" | "--"                 (. inc = false; .)
                   )                             (. if (!IsArith(des.type))
                                                    SemError("arithmetic type needed");
                                                    CodeGen.IncOrDec(inc, des.type); .)
               )
            )
```

The semantic property we need in this case can be evaluated at compile-time as follows

```
static bool IsCall(out DesType des) {
  Entry entry = Table.Find(la.val); // it must be an identifier to have got the parser this far
  des = new DesType(entry);
  return entry.kind == Kinds.Fun;
} // IsCall
```

Notice the search done on the "lookahead" token `la.val`, a little appreciated feature of Coco. Notice also that as a side effect this method (which appears at the top of the ATG file along with other "helpers") returns the `des` object corresponding to the function.

Kind of cute? The same trick could have been used had the Parva grammar allowed for value-returning functions to form part of expressions, but that can wait for another year.

Task 14 (Bonus) - Do you ever feel threatened by change?

The problem asked whether one should be allowed to change the value of the control variable associated with a *ForStatement* within the loop, as exemplified by

```
int i;  // small primes
for i in (2, 3, 5, 7, 11, 13, 17, 19, 23) {
  i = 124;  // should this be allowed to mess up the program?
  write(i, " is a small prime");
}
for i = 1 to 10 {
  for i = 1 to 10  // nested loops are allowed, but surely not this one?
    write(i);
  writeLine();
}
```

Several languages forbid "threatening" a *ForStatement* control variable. In Parva it is quite easy to detect simple cases of threatening - one simply marks the variable entry in the symbol table as unchangeable before the loop body is compiled, and resets it afterwards (why do we need to save its value first?)

```
    ForStatement<StackFrame frame>            (. int expCount;
*                                                bool canChange;
                                                 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");
                                                  CodeGen.LoadAddress(var);
*                                                 canChange = var.canChange; // save
*                                                 var.canChange = false; .)  // lock
          "in"
          ExpList<out expCount, var.type>
                                                (. CodeGen.LoadConstant(expCount);
                                                   Label loopBody = new Label(known);
                                                   CodeGen.NextControl(expCount); .)
          Statement<frame, loopExit, loopContinue>
*                                               (. var.canChange = canChange; // restore
                                                   loopContinue.Here();
                                                   CodeGen.TestFor(loopBody);
                                                   loopExit.Here();
                                                   CodeGen.ExitFor(expCount); .) .
```

ReadStatements and *AssignmentStatements* can easily check the state of the entry, as exemplified by:

```
    AssignmentOrCall                            (. int expType;
                                                   DesType des;
                                                   string name;
                                                   bool inc = true; .)
    =  (  (  IF (IsCall(out des))
             Ident<out name>
             "("                                (. CodeGen.FrameHeader(); .)
                Arguments<des>                  (. CodeGen.Call(des.entry.entryPoint); .)
             ")"
          |  Designator<out des>                (. if (des.entry.kind != Kinds.Var)
                                                     SemError("cannot assign to "
                                                              + Kinds.kindNames[des.entry.kind]);
*                                                  if (!des.canChange)
*                                                    SemError("may not alter this variable");
                                                   .)
             (   AssignOp
                 Expression<out expType>        (. if (!Assignable(des.type, expType))
                                                   SemError("incompatible types in assignment");
                                                   CodeGen.Assign(des.type); .)
               | ( "++" | "--"                  (. inc = false; .)
                 )                              (. if (!IsArith(des.type))
                                                   SemError("arithmetic type needed");
                                                   CodeGen.IncOrDec(inc, des.type); .)
             )
          )
```

In general, the problem is far more complex. Languages that support multi-function programs present far more opportunity for threatening control variables that might be passed as reference parameters, declared globally and so on, all of which make "local" checking very difficult. As we have defined the Parva *ForStatement* the control variable must be "simple" - you may have noticed that we have defined a loop by

```
ForStatement = "for" Ident "in" "(" Expression { "," Expression } ")" Statement .
```

rather than by

```
ForStatement = "for" Designator "in" "(" Expression { "," Expression } ")" Statement .
```

because protecting a simple variable like *x* is far easier than protecting a general variable like `List[i + j]`. Readers intrigued by all this might like to read up on the conditions imposed on *ForStatements* in Modula-2. But for the moment it will suffice how to prevent a control variable from being passed by reference to a function that might try to pull a dirty trick:

```
    OneArg<Entry fp>                        (. int argType;
                                               DesType des; .)
    =  (  Expression<out argType>           (. if (fp != null) {
                                                  if (!Assignable(fp.type, argType))
                                                    SemError("argument type mismatch");
                                                 if (fp.byRef)
                                                    SemError("this argument must be passed by reference");
                                               } .)
```

```
          | "ref" Designator<out des>        (. if (fp != null) {
                                                 if (!Assignable(fp.type, arg.type))
                                                   SemError("argument type mismatch");
*                                                if (!des.canChange)
*                                                  SemError("for loop control threatened");
                                                 if (!fp.byRef)
                                                   SemError("this argument must be passed by value");
                                               } .)
        ) .
```

Unfortunately, in the C family of languages more or less anything goes!

Task 15 (Bonus) - Some staff will go to any length to process a list of marks completely

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

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

```
        | "length" "(" Designator<out des>      (. if (des.entry.kind != Kinds.Var)
                                                     SemError("variable expected");
                                                   if (!IsArray(des.type))
                                                     SemError("not an array");
                                                   CodeGen.Dereference();
                                                   CodeGen.Dereference();
                                                   type = Types.intType; .)
          ")"
```

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

Task 16 (Bonus) - What if you don't like short circuit evaluation of Boolean expressions?

As you are hopefully aware, most languages implement "short-circuit semantics" in generating code for the evaluation of complex Boolean expressions. The exercise suggested that the user might be allowed to use a `$S` pragma or `-s` command line parameter to choose between code generation using short-circuit semantics or code generation using a Boolean operator approach (see the textbook, pages 22 and 365). All the opcodes you need are already in the source kit.

This is easily implemented as follows, where we have shown how the feature might be controlled by a Boolean flag set by the pragma. There is a similar change needed in *OrExp*, of course.

```
    AndExp<out int type>                     (. int type2;
                                                Label shortcircuit = new Label(!known); .)
    =  EqlExp<out type>
*      { "&&"                                (. if (shortCirc)
*                                                 CodeGen.BooleanOp(shortcircuit, CodeGen.and); .)
         EqlExp<out type2>                   (. if (!IsBool(type) || !IsBool(type2))
                                                   SemError("Boolean operands needed");
*                                               if (!shortCirc) CodeGen.BinaryOp(CodeGen.and);
                                                type = Types.boolType; .)
       }                                     (. shortcircuit.Here(); .) .
```

This is another very simple addition. I won't ask again after this, but will the students who claim that Pat Terry's Compiler Course is too difficult please stop spreading such rumours?

Task 17 (Bonus) - Generating tighter PVM code

Way back in earlier exercises we added some specialized opcodes like `LDC_1`, `LDA_2` and so on to the PVM. The problem asked for these to be used, and for a `$O` pragma or `-o` command line option to be added to the system so that these "optimized" opcodes would be used only on request (or *not* used on request - suit yourself).

The extensions to the grammar and frame files were illustrated earlier.

The code generator can respond to the pragma setting with various routines modified on the following lines (it does not seem necessary to give them all in full at this point):

```
      public static void LoadAddress(Entry var) {
      // Generates code to push address of local variable with known offset onto evaluation stack
*       if (Parser.optimize)
*         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;
*           case 4:  Emit(PVM.lda_4); break;
*           case 5:  Emit(PVM.lda_5); break;
*           default: Emit(PVM.lda); Emit(var.offset); break;
*         }
*       else {
*         Emit(PVM.lda); Emit(var.offset);
        }
      } // CodeGen.LoadAddress
```

No, I won't ask!