# RHODES UNIVERSITY

## Computer Science 301 - 2006 - Programming Language Translation

You lucky people! Here is some more free information - a complete solution to the problem posed earlier, in fact.

If one can assume that the Parva programs to be submitted to it are completely correct, a basic pretty-printer for Parva programs can be developed from the following grammar (to be found in the auxiliary kit as P1.ATG). We assume the existence of a `CodeGen` class - which essentially consists of the same methods as you saw in the earlier EBNF example.

```
import Library.*;

COMPILER Parva $NC
/* Parva level 1 grammar  - Coco/R for Java
   P.D. Terry, Rhodes University, 2006
   Pretty Printer only */

  public static String srceName;
  public static boolean
    debug = false,
    indented = true;

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

PRAGMAS
  DebugOn     = "$D+" .                 (. debug = true; .)
  DebugOff    = "$D-" .                 (. debug = false; .)

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva                                 (. String name; .)
  =  "void"                             (. CodeGen.append("void "); .)
     Ident<out name>                    (. CodeGen.append(name); .)
     "(" ")"                            (. CodeGen.append("()"); .)
     "{"                                (. CodeGen.append(" {"); CodeGen.indent(); .)
     { Statement<!indented>
     }
     WEAK "}"                           (. CodeGen.exdent(); CodeGen.newLine(); CodeGen.append("}"); .)
  .

  Statement<boolean indented>
  =     Block
      | ";"                             (. CodeGen.append(";"); .)
                                        (. if (indented) CodeGen.indent(); CodeGen.newLine(); .)
       (   Assignment
         | ConstDeclarations
         | VarDeclarations
         | IfStatement
         | WhileStatement
         | DoWhileStatement
         | ForStatement
         | BreakStatement
         | ContinueStatement
         | HaltStatement
         | ReturnStatement
         | ReadStatement
         | WriteStatement
       )                                (. if (indented) CodeGen.exdent(); .)
  .
```

```
Block
=
  "{"                                (. CodeGen.append(" {"); CodeGen.indent(); .)
   { Statement<!indented>
   }
  WEAK "}"                           (. CodeGen.exdent(); CodeGen.newLine(); CodeGen.append("}"); .)
.

ConstDeclarations
= "const"                            (. CodeGen.append("const "); .)
  OneConst
  { WEAK ","                         (. CodeGen.append(", "); .)
    OneConst
  } WEAK ";"                         (. CodeGen.append(";"); .)
.

OneConst                             (. String name; .)
=  Ident<out name>                   (. CodeGen.append(name); .)
   "="                               (. CodeGen.append(" = "); .)
   Constant .

Constant                             (. String value; .)
=   IntConst<out value>              (. CodeGen.append(value); .)
  | CharConst<out value>             (. CodeGen.append(value); .)
  | "true"                           (. CodeGen.append("true");  .)
  | "false"                          (. CodeGen.append("false"); .)
  | "null"                           (. CodeGen.append("null");  .)
.

VarDeclarations
=  Type                              (. CodeGen.append(" "); .)
   OneVar
   { WEAK ","                        (. CodeGen.append(", "); .)
     OneVar
   }
   WEAK ";"                          (. CodeGen.append(";"); .)
.

Type
=  BasicType
   [ "[]"                            (. CodeGen.append("[]"); .)
   ] .

BasicType
=   "int"                            (. CodeGen.append("int");  .)
  | "bool"                           (. CodeGen.append("bool"); .)
  | "char"                           (. CodeGen.append("char"); .)
.

OneVar                               (. String name; .)
=  Ident<out name>                   (. CodeGen.append(name); .)
   [ "="                             (. CodeGen.append(" = "); .)
     Expression
   ] .

Assignment
=  Designator
   (  "="                            (. CodeGen.append(" = "); .)
      Expression
    | "++"                           (. CodeGen.append("++"); .)
    | "--"                           (. CodeGen.append("--"); .)
   )
   WEAK ";"                          (. CodeGen.append(";"); .)
.

Designator                           (. String name; .)
= Ident<out name>                    (. CodeGen.append(name); .)
   [  "["                            (. CodeGen.append("["); .)
         Expression
      "]"                            (. CodeGen.append("]"); .)
   ] .

IfStatement
= "if"                               (. CodeGen.append("if "); .)
  "("                                (. CodeGen.append("("); .)
  Condition
  ")"                                (. CodeGen.append(")"); .)
  Statement<indented>
  [ "else"                           (. CodeGen.newLine(); CodeGen.append("else "); .)
    Statement<indented>
  ] .
```

```
WhileStatement
=  "while"                          (. CodeGen.append("while "); .)
   "("                              (. CodeGen.append("("); .)
   Condition
   ")"                              (. CodeGen.append(")"); .)
   Statement<indented>
.

DoWhileStatement
=  "do"                             (. CodeGen.append("do"); .)
     Statement<indented>
   WEAK "while"                     (. CodeGen.newLine(); CodeGen.append("while "); .)
   "("                              (. CodeGen.append("("); .)
   Condition
   ")"                              (. CodeGen.append(")"); .)
   WEAK ";"                         (. CodeGen.append(";"); .)
.

ForStatement                        (. String name; .)
=  "for"                            (. CodeGen.append("for "); .)
   Ident<out name>                  (. CodeGen.append(name); .)
   "="                              (. CodeGen.append(" = "); .)
   Expression
   (   "to"                         (. CodeGen.append(" to "); .)
     | "downto"                     (. CodeGen.append(" downto "); .)
   )
   Expression
   Statement<indented>
.

BreakStatement
=  "break"                          (. CodeGen.append("break"); .)
   WEAK ";"                         (. CodeGen.append(";"); .)
.

ContinueStatement
=  "continue"                       (. CodeGen.append("continue"); .)
   WEAK ";"                         (. CodeGen.append(";"); .)
.

HaltStatement
=  "halt"                           (. CodeGen.append("halt"); .)
   WEAK ";"                         (. CodeGen.append(";"); .)
.

ReturnStatement
=  "return"                         (. CodeGen.append("return"); .)
   WEAK ";"                         (. CodeGen.append(";"); .)
.

ReadStatement
= "read"                            (. CodeGen.append("read"); .)
  "("                               (. CodeGen.append("("); .)
  ReadElement
  { WEAK ","                        (. CodeGen.append(", "); .)
    ReadElement
  }
  ")"                               (. CodeGen.append(")"); .)
  WEAK ";"                          (. CodeGen.append(";"); .)
.

ReadElement                         (. String str; .)
= (   StringConst<out str>          (. CodeGen.append(str); .)
    | Designator
  )
.

WriteStatement
= "write"                           (. CodeGen.append("write"); .)
  "("                               (. CodeGen.append("("); .)
  WriteElement
  { WEAK ","                        (. CodeGen.append(", "); .)
    WriteElement
  }
  ")"                               (. CodeGen.append(")"); .)
  WEAK ";"                          (. CodeGen.append(";"); .)
.
```

```
WriteElement                          (. String str; .)
= (   StringConst<out str>            (. CodeGen.append(str); .)
    | Expression
  )
.

Condition
=  Expression
.

Expression
=  AndExp
   { "||"                             (. CodeGen.append(" || "); .)
     AndExp
   }
.

AndExp
=  EqlExp
   { "&&"                             (. CodeGen.append(" && "); .)
     EqlExp
   }
.

EqlExp
=  RelExp
   { EqualOp
     RelExp
   } .

RelExp
=  AddExp
   [ RelOp
     AddExp
   ] .

AddExp
=  MultExp
   { AddOp
     MultExp
   } .

MultExp
=  Factor
   { MulOp
     Factor
   } .

Factor
=    Primary
   | "+"                              (. CodeGen.append(" +"); .)
     Factor
   | "-"                              (. CodeGen.append(" -"); .)
     Factor
   | "!"                              (. CodeGen.append(" !"); .)
     Factor
.

Primary
=    Designator
   | Constant
   | "new"                            (. CodeGen.append("new "); .)
     BasicType
     "["                              (. CodeGen.append("["); .)
     Expression
     "]"                              (. CodeGen.append("]"); .)
   | "("                              (. CodeGen.append("("); .)
     (   "char" ")"                   (. CodeGen.append("char)"); .)
         Factor
       | "int" ")"                    (. CodeGen.append("int)"); .)
         Factor
       | Expression
         ")"                          (. CodeGen.append(")"); .)
     )
.

AddOp
=    "+"                              (. CodeGen.append(" + "); .)
   | "-"                              (. CodeGen.append(" - "); .)
.
```

```
  MulOp
  =    "*"                          (. CodeGen.append(" * "); .)
     | "/"                          (. CodeGen.append(" / "); .)
     | "%"                          (. CodeGen.append(" % "); .)
  .

  EqualOp
  =    "=="                         (. CodeGen.append(" == "); .)
     | "!="                         (. CodeGen.append(" != "); .)
  .

  RelOp
  =    "<"                          (. CodeGen.append(" < "); .)
     | "<="                         (. CodeGen.append(" <= "); .)
     | ">"                          (. CodeGen.append(" > "); .)
     | ">="                         (. CodeGen.append(" >= "); .)
  .

  Ident<out String name>
  =  identifier                     (. name = token.val; .)
  .

  StringConst<out String name>
  = stringLit                       (. name = token.val; .)
  .

  CharConst<out String name>
  =  charLit                        (. name = token.val; .)
  .

  IntConst<out String name>
  =  number                         (. name = token.val; .)
  .

END Parva.
```

About the only thing to draw to your attention is the way in which the `indented` argument to the `Statement` production is used to handle the indentation of the subsidiary statements found in *if, while* and *do-while* statements.

"There is always a better way".

One can do considerably better than this, if one is prepared to do some semantic checking as well, in the spirit of that described in the textbook in chapter 12. Here is a rather longer grammar, with further actions added like those you saw in the final practical. This system not only does pretty-printing, but also checks for type mismatches, undeclared variables, break statements not embedded in loops and all those awful blunders that programmers sometimes make.

```
import Library.*;

COMPILER Parva $NC
/* Parva level 1 grammar  - Coco/R for Java
   P.D. Terry, Rhodes University, 2006
   Java operator precedences
   Includes character type
   Pretty Printer with semantic checking
   No functions and no optimization */

  public static String srceName;

  public static boolean
    debug = false,
    indented = true;

  static int loopLevel = 0;                     // = 0 outside of loops, > 0 inside loops

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

  static boolean isBool(int type) {
    return type == Entry.boolType || type == Entry.noType;
  }

  static boolean isRef(int type) {
    return (type % 2) == 1;
  }
```

```
  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;
  }

  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;
  }

/*---------------------------------------------------------------------------*/

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

PRAGMAS
  DebugOn     = "$D+" .                 (. debug = true; .)
  DebugOff    = "$D-" .                 (. debug = false; .)

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva
  =  "void"                             (. Entry program = new Entry();
                                           CodeGen.append("void "); .)
     Ident<out program.name>            (. CodeGen.append(program.name); .)
     "(" ")"                            (. CodeGen.append("()");
                                           program.kind = Entry.Fun;
                                           program.type = Entry.voidType;
                                           Table.insert(program);
                                           Table.openScope(); .)
     "{"                                (. CodeGen.append(" {"); CodeGen.indent(); .)
     { Statement<!indented>
     }
     WEAK "}"                           (. CodeGen.exdent(); CodeGen.newLine(); CodeGen.append("}");
                                           if (debug) Table.printTable(OutFile.StdOut);
                                           Table.closeScope(); .)
  .

  Statement<boolean indented>
  =     Block
      | ";"                             (. CodeGen.append(";"); .)
      |                                 (. if (indented) CodeGen.indent(); CodeGen.newLine(); .)
       (   Assignment
         | ConstDeclarations
         | VarDeclarations
         | IfStatement
         | WhileStatement
         | DoWhileStatement
         | ForStatement
         | BreakStatement
         | ContinueStatement
         | HaltStatement
         | ReturnStatement
         | ReadStatement
         | WriteStatement
       )                                (. if (indented) CodeGen.exdent(); .)
  .
```

```
Block
=                                   (. Table.openScope(); .)
  "{"                               (. CodeGen.append(" {"); CodeGen.indent(); .)
   { Statement<!indented>
   }
  WEAK "}"                          (. CodeGen.exdent(); CodeGen.newLine(); CodeGen.append("}");
                                       if (debug) Table.printTable(OutFile.StdOut);
                                       Table.closeScope(); .)
.

ConstDeclarations
= "const"                           (. CodeGen.append("const "); .)
  OneConst
  { WEAK ","                        (. CodeGen.append(", "); .)
    OneConst
  } WEAK ";"                        (. CodeGen.append(";"); .)
.

OneConst                            (. Entry constant = new Entry();
                                       ConstRec con; .)
=  Ident<out constant.name>         (. constant.kind = Entry.Con; .)
   "=" Constant<out con>            (. CodeGen.append(constant.name + " = " + con.name);
                                       constant.type = con.type;
                                       Table.insert(constant); .) .

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

VarDeclarations                     (. int type; .)
=  Type<out type>                   (. CodeGen.append(" "); .)
   OneVar<type>
   { WEAK ","                       (. CodeGen.append(", "); .)
     OneVar<type>
   }
   WEAK ";"                         (. CodeGen.append(";"); .)
.

Type<out int type>
=  BasicType<out type>
   [ "[]"                           (. CodeGen.append("[]");
                                       if (type != Entry.noType) type++; .)
   ] .

BasicType<out int type>             (. type = Entry.noType; .)
=   "int"                           (. CodeGen.append("int");     type = Entry.intType; .)
  | "bool"                          (. CodeGen.append("bool");    type = Entry.boolType; .)
  | "char"                          (. CodeGen.append("char");    type = Entry.charType; .)
.

OneVar<int type>                    (. int expType; .)
=                                   (. Entry var = new Entry(); .)
   Ident<out var.name>              (. CodeGen.append(var.name);
                                       var.kind = Entry.Var;
                                       var.type = type; .)
   [ "="                            (. CodeGen.append(" = "); .)
     Expression<out expType>        (. if (!assignable(var.type, expType))
                                         SemError("incompatible types in assignment"); .)
   ]                                (. Table.insert(var); .)
.

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

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

IfStatement
= "if"                                (. CodeGen.append("if "); .)
  "("                                 (. CodeGen.append("("); .)
  Condition
  ")"                                 (. CodeGen.append(")"); .)
  Statement<indented>
  [ "else"                            (. CodeGen.newLine(); CodeGen.append("else "); .)
    Statement<indented>
  ]
.

WhileStatement                        (. loopLevel++; .)
=  "while"                            (. CodeGen.append("while "); .)
   "("                                (. CodeGen.append("("); .)
   Condition
   ")"                                (. CodeGen.append(")"); .)
   Statement<indented>                (. loopLevel--; .)
.

DoWhileStatement                      (. loopLevel++; .)
=  "do"                               (. CodeGen.append("do"); .)
     Statement<indented>
   WEAK "while"                       (. CodeGen.newLine(); CodeGen.append("while "); .)
   "("                                (. CodeGen.append("("); .)
   Condition
   ")"                                (. CodeGen.append(")"); .)
   WEAK ";"                           (. CodeGen.append(";");
                                         loopLevel--; .)
.

ForStatement                          (. boolean up = true;
                                         DesType des;
                                         String name;
                                         int expType;
                                         loopLevel++; .)
=  "for"                              (. CodeGen.append("for "); .)

   Ident<out name>                    (. CodeGen.append(name);
                                         Entry entry = Table.find(name);
                                         if (!entry.declared)
                                           SemError("undeclared identifier");
                                         des = new DesType(entry);
                                         if (isRef(des.type) || des.entry.kind != Entry.Var)
                                           SemError("illegal control variable"); .)
   "="                                (. CodeGen.append(" = "); .)
   Expression<out expType>            (. if (!assignable(des.type, expType))
                                           SemError("incompatible with control variable"); .)
   (   "to"                           (. CodeGen.append(" to "); .)
     | "downto"                       (. CodeGen.append(" downto ");
                                         up = false; .)
   )
   Expression<out expType>            (. if (!assignable(des.type, expType))
                                           SemError("incompatible with control variable"); .)
   Statement<indented>                (. loopLevel--; .)
.

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

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

HaltStatement
=  "halt"                              (. CodeGen.append("halt"); .)
   WEAK ";"                            (. CodeGen.append(";"); .)
.

ReturnStatement
=  "return"                            (. CodeGen.append("return"); .)
   WEAK ";"                            (. CodeGen.append(";"); .)
.

ReadStatement
= "read"                               (. CodeGen.append("read"); .)
  "("                                  (. CodeGen.append("("); .)
  ReadElement
  { WEAK ","                           (. CodeGen.append(", "); .)
    ReadElement
  }
  ")"                                  (. CodeGen.append(")"); .)
  WEAK ";"                             (. CodeGen.append(";"); .)
.

ReadElement                            (. String str;
                                          DesType des; .)
= (   StringConst<out str>             (. CodeGen.append(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:
                                              break;
                                            default:
                                              SemError("cannot read this type"); break;
                                          } .)
  )
.

WriteStatement
= "write"                              (. CodeGen.append("write"); .)
  "("                                  (. CodeGen.append("("); .)
  WriteElement
  { WEAK ","                           (. CodeGen.append(", "); .)
    WriteElement
  }
  ")"                                  (. CodeGen.append(")"); .)
  WEAK ";"                             (. CodeGen.append(";"); .)
.

WriteElement                           (. int expType;
                                          String str; .)
= (   StringConst<out str>             (. CodeGen.append(str); .)
    | Expression<out expType>          (. switch (expType) {
                                            case Entry.intType:
                                            case Entry.boolType:
                                            case Entry.charType:
                                              break;
                                            default:
                                              SemError("cannot write this type"); break;
                                          } .)
  ) .

Condition                              (. int type; .)
=  Expression<out type>                (. if (!isBool(type))
                                            SemError("boolean expression needed"); .)
.

Expression<out int type>               (. int type2; .)
=  AndExp<out type>
   { "||"                              (. CodeGen.append(" || "); .)
     AndExp<out type2>                 (. if (!isBool(type) || !isBool(type2))
                                            SemError("Boolean operands needed");
                                          type = Entry.boolType; .)
   }
.
```

```
AndExp<out int type>                (. int type2; .)
=  EqlExp<out type>
   { "&&"                           (. CodeGen.append(" && "); .)
     EqlExp<out type2>              (. if (!isBool(type) || !isBool(type2))
                                         SemError("Boolean operands needed");
                                       type = Entry.boolType; .)
   }
.

EqlExp<out int type>                (. int type2; .)
=  RelExp<out type>
   { EqualOp
     RelExp<out type2>              (. if (!compatible(type, type2))
                                         SemError("incomparable operands");
                                       type = Entry.boolType; .)
   } .

RelExp<out int type>                (. int type2; .)
=  AddExp<out type>
   [ RelOp
     AddExp<out type2>              (. if (!isArith(type) || !isArith(type2))
                                         SemError("incomparable operands");
                                       type = Entry.boolType; .)
   ] .

AddExp<out int type>                (. int type2; .)
=  MultExp<out type>
   { AddOp
     MultExp<out type2>             (. if (!isArith(type) || !isArith(type2)) {
                                         SemError("arithmetic operands needed");
                                         type = Entry.noType;
                                       }
                                       else type = Entry.intType; .)
   } .

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

Factor<out int type>                (. type = Entry.noType; .)
=    Primary<out type>
   | "+"                            (. CodeGen.append(" +"); .)
     Factor<out type>               (. if (!isArith(type)) {
                                         SemError("arithmetic operand needed");
                                         type = Entry.noType;
                                       }
                                       else type = Entry.intType; .)
   | "-"                            (. CodeGen.append(" -"); .)
     Factor<out type>               (. if (!isArith(type)) {
                                         SemError("arithmetic operand needed");
                                         type = Entry.noType;
                                       }
                                       else type = Entry.intType; .)
   | "!"                            (. CodeGen.append(" !"); .)
     Factor<out type>               (. if (!isBool(type))
                                         SemError("Boolean operand needed");
                                       type = Entry.boolType; .)
.

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:
                                         case Entry.Con:
                                           break;
                                         default:
                                           SemError("wrong kind of identifier");
                                           break;
                                       } .)
   | Constant<out con>              (. CodeGen.append(con.name); type = con.type; .)
```

```
      | "new"                          (. CodeGen.append("new "); .)
        BasicType<out type>            (. type++; .)
        "["                            (. CodeGen.append("["); .)
        Expression<out size>           (. if (!isArith(size))
                                            SemError("array size must be integer"); .)
        "]"                            (. CodeGen.append("]"); .)
      | "("                            (. CodeGen.append("("); .)
        (   "char" ")"                 (. CodeGen.append("char)"); .)
            Factor<out type>           (. if (!isArith(type))
                                            SemError("invalid cast");
                                          else type = Entry.charType; .)
          | "int" ")"                  (. CodeGen.append("int)"); .)
            Factor<out type>           (. if (!isArith(type))
                                            SemError("invalid cast");
                                          else type = Entry.intType; .)
          | Expression<out type>
            ")"                        (. CodeGen.append(")"); .)
        )
.

AddOp
=     "+"                              (. CodeGen.append(" + "); .)
    | "-"                              (. CodeGen.append(" - "); .)
.

MulOp
=     "*"                              (. CodeGen.append(" * "); .)
    | "/"                              (. CodeGen.append(" / "); .)
    | "%"                              (. CodeGen.append(" % "); .)
.

EqualOp
=     "=="                             (. CodeGen.append(" == "); .)
    | "!="                             (. CodeGen.append(" != "); .)
.

RelOp
=     "<"                              (. CodeGen.append(" < "); .)
    | "<="                             (. CodeGen.append(" <= "); .)
    | ">"                              (. CodeGen.append(" > "); .)
    | ">="                             (. CodeGen.append(" >= "); .)
.

Ident<out String name>
=  identifier                          (. name = token.val; .)
.

StringConst<out String name>
= stringLit                            (. name = token.val; .)
.

CharConst<out String name>
=  charLit                             (. name = token.val; .)
.

IntConst<out String name>
=  number                              (. name = token.val; .)
.

END Parva.
```

Much of this code should be familiar from your earlier practical sessions. To prepare yourself to answer Section B of the examination tomorrow, you are encouraged to study it in depth. Questions in Section B will probe this understanding, and you might be called on to make some modifications and extensions.

That gives me a lot of scope, does it not? Here are some things to think about. Last year's exam required people to produce HTML output - how might you modify this system to do that, highlighting the key words in some colour? Or suppose one wanted to choose the indentation level by using a pragma, rather than using a fixed value of 2? Or suppose somebody had the misguided idea that he wanted to convert Parva programs from a syntax where the key words were given in lower case to one where they were given in UPPER CASE? Or suppose one wanted to generate a pretty version of a program, adding line numbers as in the example below, possibly followed by a cross reference listing of all the identifiers?

```
/*   1 */  void main() {
/*   2 */    int b, c, d;
/*   3 */    b = 10;
/*   4 */    while (b > 0) {
/*   5 */      write("b = ", b);
/*   6 */      b--;
/*   7 */      if (b == 4)
/*   8 */        write(" hit four!\n");
/*   9 */      else
/*  10 */        write(" on we go\n");
/*  11 */    }
/*  12 */  }
```

As before, you are quite at liberty to experiment with the system and to discuss it with your prac partners, but not with staff or demonstrators.

I have a vivid imagination.  Do you?

Have fun!

*You will receive a copy of the attributed grammar tomorrow, and a copy in machine-readable form.*