RHODES UNIVERSITY

November Examinations - 2006

Computer Science 301 - Paper 2 - Solutions

Examiners:
Prof P.D. Terry
Prof J.H. Greyling

Time 3 hours
Marks 180
Pages 11 (please check!)

Answer all questions. Answers may be written in any medium except red ink.

(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included an augmented version of "Section C" - a request to develop a simple Parva pretty-printer. 18 hours before the examination a complete grammar and other support files for building a more sophisticated pretty-printer were released along with explanatory text - summarized here as "Section D". During the examination, candidates were given machine executable versions of the Coco/R compiler generator, the files needed to build the pretty-printer, access to a computer, and machine readable copies of the questions.)

Section A [90 marks]

A1. (a) What distinguishes a *context free grammar* from a *context sensitive grammar*? [2 marks]

In general, grammar productions take the form

$$\alpha \rightarrow \beta \qquad \text{with} \quad \alpha \in (N \cup T)^* N (N \cup T)^* \; ; \quad \beta \in (N \cup T)^*$$

where N is the set of non-terminals and T is the set of terminals. In context-free grammars $\alpha \in N$ *(the left side of each production must consist of a single non-terminal); for a grammar to be context-sensitive there must be at least one production for which* α *is comprised of two or more tokens. Note the importance of the phrase "at least one production".*

(b) The syntax of many programming languages is described by a context free grammar, and yet there are properties of most programming languages that are context sensitive. Mention one such property, and indicate briefly how this context sensitivity is handled in practical compilers. [4 marks]

There are plenty of examples to choose from - such as

- *variables must be declared before they are used*
- *the number of formal and actual arguments for functions must agree*
- *expressions used to control if and while statements must be of Boolean type*
- *values assigned to variables must be of the appropriate type*
- *break statements may only be used in the context of a loop or switch statement*
- *a nice one suggested by this paper, question A7 - in a multiple assignment statement ther must be as many "designators" as there are "expressions".*

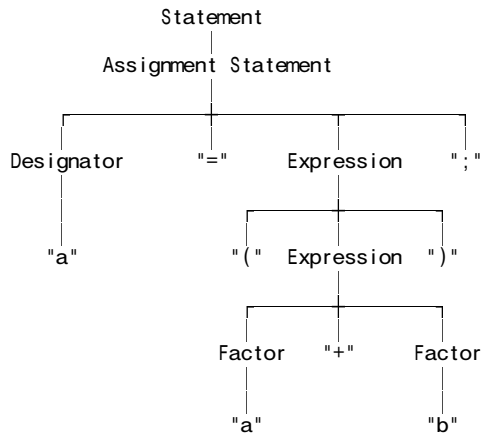
Many of these are usually handled by making use of a "symbol table" in which the various properties of the items denoted by identifiers are recorded, or by checking one count against another.

A2. What distinguishes a *concrete syntax tree* from an *abstract syntax tree*? Illustrate your answer by drawing each form of tree for the simple Java statement [6 marks]

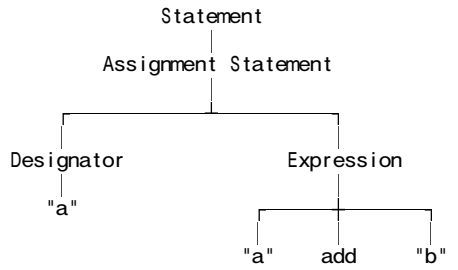
```
a = ( b + c ) ;
```

A concrete syntax tree incorporates all the tokens used in deriving the parse tree, while an abstract syntax tree retains only the information needed later to analyse the semantics completely:

Concrete



Abstract



A3. Describe briefly what you understand by the following two properties of variables in block structured imperative programming languages - firstly, their *scope* and, secondly, their *existence*. [4 marks]

The scope of an identifier in a block structured language is the area of code in which it can be recognised - typically the block in which it is declared (and any blocks nested within that block, subject to rules that might allow redeclaration on condition that the most recent declaration applies). Existence refers to the length of time at run time for which the variable has storage allocated to it - typically using a stack based runtime structure ensuring that such storage is allocated and deallocated as blocks are activated and deactivated.

A4. (a) What distinguishes a *native code compiler* from an *interpretive compiler*? [2 marks]

A native code compiler generates machine level object code, typically for the same machine that is executing the compiler itself. An interpretive compiler generates intermediate level code, typically for a virtual machine whose operation can be emulated by a suitable interpreter for such code.

(b) Suggest one property of native code compilation that is claimed to offer an advantage over interpretive compilation, and also one property of interpretive compilation that is claimed to offer an advantage over native code compilation. [2 marks]

Native code compilers should produce object code that can be executed at the full speed of the host machine, typically one or two orders of magnitude faster than an interpreter can execute code for a virtual machine. Interpretive compilers on the other hand are much more easily developed, and can be made highly portable (all that is needed is a suitable interpreter for the virtual code).

(c) What do you understand by the technique known by the acronym JIT, and what systems known to you incorporate this technique? [4 marks]

JIT stands for "just in time" and refers to the technique frequently used by Java and .NET systems of completing the translation of code for a virtual machine or interpreter into native machine code whenever and only when it is needed.

A5. *(Attributed grammars)* The following familiar Cocol grammar describes a set of EBNF productions (of the form found in the `PRODUCTIONS` section of the grammar itself).

```
COMPILER EBNF $CN
/* Describe a set of EBNF productions
   P.D. Terry, Rhodes University, 2006 */

CHARACTERS
  eol         = CHR(10) .
  space       = CHR(32) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  lowline     = "_" .
  printable   = ANY - control .
  nonSpace    = printable - space .
  noquote1    = ANY - "'" - control .
  noquote2    = ANY - '"' - control .

TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .

IGNORE control

PRODUCTIONS
  EBNF        = { Production } EOF .
  Production  = nonterminal "=" Expression  "." .
  Expression  = Term { "|" Term } .
  Term        = Factor { Factor } .
  Factor      =   nonterminal | terminal | "(" Expression ")"
                | "[" Expression "]" | "{" Expression "}" .
END EBNF.
```

(a) In the original notation known as BNF, productions took a form exemplified by

```
<nonterminal> ::= ε  |  terminal1  | ( <another nonterminal> | terminal2 ) + <something>
```

The notation allowed the use of an explicit ε. [] brackets and {} braces were not used (although () parentheses were allowed). Non-terminals and terminals were distinguished by the presence or absence of < > angle brackets, and a production was terminated at the end of a line.

Describe a set of BNF productions using Cocol (simply modify the grammar above). [15 marks]

We have to redefine the form that terminals and non-terminals take in the TOKENS section. This is the trickiest part to get right, but something like this had been discussed in tutorials and tests. The transformation of the PRODUCTIONS section is straightforward. Note where the null option ε is introduced!

```
COMPILER BNF $CN
/* Describe a set of EBNF productions
  P.D. Terry, Rhodes University, 2006 */

CHARACTERS
  eol         = CHR(10) .
  space       = CHR(32) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  lowline     = "_" .
  printable   = ANY - control .
  nonSpace    = printable - space .
  startTerm   = printable - "<" .

TOKENS
  nonterminal = "<" letter { letter | lowline | digit | space } ">" .
  terminal    = startTerm { nonSpace } | "<"  | '"("' | '")"' .
  EOL         = eol .

IGNORE control - eol

PRODUCTIONS
  BNF         = { Production } EOF .
  Production  = nonterminal "::=" Expression EOL .
  Expression  = [ "ε" "|" ] Term { "|" Term } .
  Term        = Factor { Factor } .
  Factor      = nonterminal | terminal | "(" Expression ")" .
END BNF.
```

This question was badly understood or misread by too many students. You were asked to describe BNF notation using Cocol, not to write a description of EBNF using BNF - and certainly were not being asked to write a grammar that had eliminated all the meta-brackets.

(b) Assume that you have available a suitable scanner method called `getSym` that can recognize the terminals of BNF and classify them appropriately as members of the following enumeration

```
EOFSym, noSym, EOLSym, termSym, nontermSym, definedBySym,
epsilonSym, barSym, lParenSym, rParenSym
```

Develop a hand-crafted recursive descent parser for recognizing a set of BNF productions based on your description in (a). *(Your parser can take drastic action if an error is detected. Simply call methods like* `accept` *and* `abort` *to produce appropriate error messages and then terminate parsing. You are not required to write any code to implement the* `getSym`, `accept` *or* `abort` *methods.)* [15 marks]

What was expected was code on the following lines:

```
static SymSet FirstFactor = new SymSet(new int[] {lParenSym, termSym, nontermSym});

static void BNF() {
  while (sym == nontermSym) {
    Production();
  }
  accept(EOFSym, "EOF expected");
}

static void Production() {
  getSym();
  accept(definedBySym, "::= expected");
  Expression();
  accept(EOLSym, "productions must be terminated by end-of-line");
}

static void Expression() {
  if (sym == epsilonSym) {
    getSym();
    accept(barSym, "| expected");
  }
  Term();
  while (sym == barSym) {
    getSym(); Term();
  }
}

static void Term() {
  Factor();
  while (FirstFactor.contains(sym)) Factor();
}

static void Factor () {
  switch (sym) {
    case nontermSym :
    case termSym :
      getSym();
      break;
    case lParenSym :
      getSym(); Expression(); accept(rParenSym, ") expected");
      break;
    default:
      abort("invalid start to Factor");
      break;
  }
}
```

A6. *(Grammars)* By now you should be familiar with RPN or "Reverse Polish Notation" as a notation that can describe expressions without the need for parentheses. The notation eliminates parentheses by using "postfix" operators after the operands. To evaluate such expressions one uses a stack architecture, such as formed the basis of the PVM machine studied in the course. Examples of RPN expressions are:

`3 4 +`	- equivalent to	`3 + 4`
`3 4 5 + *`	- equivalent to	`3 * (4 + 5)`

In many cases an operator is taken to be "binary" - applied to the two preceding operands - but the notation is sometimes extended to incorporate "unary" operators - applied to one preceding operand:

```
4 sqrt          - equivalent to   sqrt(4)
5 -             - equivalent to   -5
```

The following represent two attempts to write a grammar for an RPN expression:

```
(G1)    RPN     =    RPN RPN binOp
                   | RPN unaryOp
                   | number .
        binOp   =  "+" | "-" | "*" | "/" .
        unaryOp =  "-" | "sqrt" .

(G2)    RPN     =  number REST .
        REST    =  [ number REST binOp REST | unaryOp ].
        binOp   =  "+" | "-" | "*" | "/" .
        unaryOp =  "-" | "sqrt" .
```

(a) What do you understand by an *ambiguous grammar* and what do you understand by *equivalent grammars*? [4 marks]

An ambiguous grammar is one for which at least one sentence can be derived in more than one way, that is, for which the parse tree is not unique.

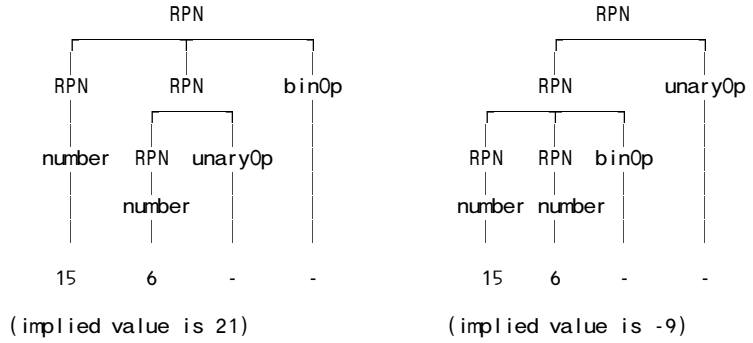
Grammars are equivalent if they derive exactly the same set of sentences (not necessarily using the same set of sentential forms or parsertrees).

(b) Using the expression

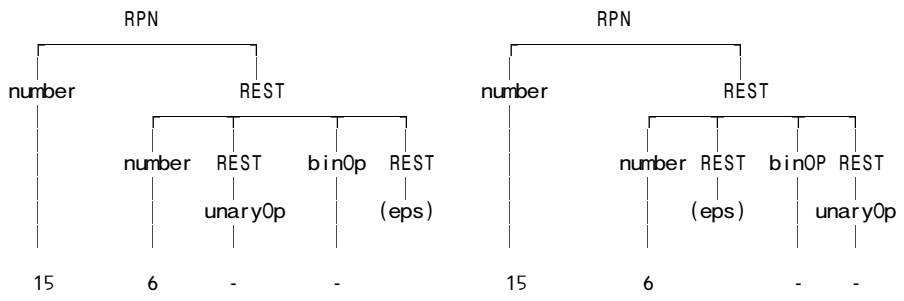
```
15  6  -  -
```

as an example, and by drawing appropriate parse trees, demonstrate that both of the grammars above are ambiguous. [6 marks]

Using grammar G1:



Using grammar G2:



(c) Analyse each of these grammars to check whether they conform to the LL(1) conditions, explaining quite clearly (if they do not!) where the rules are broken. [6 marks]

G1 is left recursive and this cannot be an LL(1) grammar. There are two alternatives for the right side of the production for RPN that both start with RPN, so Rule 1 is broken

*For G2, REST is nullable. First(REST) is { number, sqrt, - } while Follow(REST) is { +, -, /, * } so Rule 2 is broken.*

A7. (Code generation) A BYT (Bright Young Thing) has been nibbling away at writing extensions to her first Parva compiler, while learning the Python language at the same time. She has been impressed by a Python feature which allows one to write multiple assignments into a single statement, as exemplified by

```
A, B = X + Y, 3 * List[6];
A, B = B, A;                // exchange A and B
A, B = X + Y, 3, 5;         // incorrect
```

which she correctly realises can be described by the context-free production

```
Assignment = Designator { "," Designator } "=" Expression { "," Expression } ";"
```

The Parva attributed grammar that she has been given deals with single, simple integer assignments only:

```
Assignment                      (. DesType des; .)
=  Designator<out des>          (. if (des.entry.kind != Entry.Var)
                                     SemError("invalid assignment"); .)
   "="
   Expression                   (. CodeGen.assign(); .)
   ";" .
```

where the `CodeGen.assign()` method generates a code that is matched in the interpreter by a `case` arm reading

```
case PVM.sto:
  //   store value at top of stack on address at second on stack
  mem[mem[cpu.sp + 1]] = mem[cpu.sp];

  //   bump stack pointer
  cpu.sp = cpu.sp + 2;
  break;
```

Suggest, in as much detail as time will allow, how the `Assignment` production and the interpreter would need to be changed to support this language extension. [20 marks]

To ensure that there are as many expressions as designators it is easiest to count both, and then make a simple check when the terminating semicolon is identified. This count can be used as an argument to a new `PVM.sto` *opcode:*

```
        Assignment                  (. DesType des;
                                       int desCount = 1, expCount = 1; .)
***     =  Designator<out des>      (. if (des.entry.kind != Entry.Var)
                                       SemError("invalid assignment"); .)
***        { "," Designator<out des>(. if (des.entry.kind != Entry.Var)
***                                   SemError("invalid assignment");
***                                 desCount++; .)
***        }
           "="
           Expression
***        { "," Expression         (. expCount++; .)
***        }                        (. if (expCount != desCount)
***                                   SemError("left and right counts disagree");
***                                 CodeGen.assign(desCount); .)
           ";" .
```

where the action of the new `PVM.sto` *opcode is to loop through the appropriate number of assignments and then to pop 2n values off the operand stack:*

```
        case PVM.sto:
          //   store n values at top of stack on address at second on stack
          int n = Next();
          for (int i = n - 1; i >= 0; i--)
            mem[mem[cpu.sp + n + i] = mem[cpu.sp + i];

          //   bump stack pointer
          cpu.sp = cpu.sp + 2 * n;
          break;
```

Section B [90 marks]

Please note that there is no obligation to produce a machine readable solution for this section. Coco/R and other files are provided so that you can enhance, refine, or test your solution if you desire. If you choose to produce a machine readable solution, you should create a working directory, unpack EXAM.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

Regular readers of the Journal of Anxiety and Stress will have realized that at about this time every year the Department of Computer Science has a last minute crisis, and 2006 is proving to be no exception. Believe it or not, the Department is due to mount a first year practical examination this afternoon, and have managed to lose every single copy of the Parva compiler needed by the students!

What should one do in such an emergency? Dial 911? Dial the Dean of Science?

If you try the latter he will put on his wicked smile and ask: "Are you not one of those people who spent a happy weekend developing a pretty-printer for Parva?" And when you admit that you are, he will smile even more broadly and then say: "So what's the problem? Simply modify the pretty-printer so that it produces Java output in place of Parva output, and then the first year students will never notice the difference - they will automagically be able to convert their Parva programs to Java and then compile them with the Java compiler instead."

As usual, he's right! Parva programs are really very similar to Java ones, except for a few simple differences, as the following example will illustrate:

Parva: (demo.pav)

```
                              // A Parva program has a single void method
  void main () {
    int year, yourAge;
                              // Parva uses the keyword bool
    bool olderThanMe;
                              // Parva constants are defined like this
    const myAge = 61, overTheHill = true;
                              // Parva has a multiple argument read statement
    read("How old are you? ", yourAge);
    if (yourAge < 0 || yourAge > 100) {
      write("I simply do not believe you!");
                              // Parva has a halt statement
      halt;
    }
    bool olderThanYou = myAge > yourAge;
    read("Do you think you are older than I am? ", olderThanMe);
                              // Parva has a multiple argument write statement
    write("Your claim that you are older than me is", olderThanMe != olderThanYou);
                              // Parva has a Pascal-like for loop
    for year = yourAge to myAge
      write("!");
    write(" - I am", myAge);
  }
```

Java equivalent: (demo.java)

```
                              // A simple Java program has a single class
                              // and usually has to import library classes
  import Library.*;

  class demo {
                              // A simple Java program has a standard main method
    public static void main(String[] args) {
      int year, yourAge;
                              // Java uses the keyword boolean
      boolean olderThanMe;
```

```
                              // Java constants are defined like this
      final int myAge = 61;
      final boolean overTheHill = true;
                              // Java might use typed methods from the IO library for input
      { IO.write("How old are you? "); yourAge = IO.readInt(); }
      if (yourAge < 0 || yourAge > 100) {
        { IO.write("I simply do not believe you!"); }
                              // Java has the equivalent of a halt statement
        System.exit(0);
      }
      boolean olderThanYou = myAge > yourAge;
      { IO.write("Do you think you are older than I am? "); olderThanMe = IO.readBool(); }
                              // Java might use single argument methods from the IO library for output
      { IO.write("Your claim that you are older than me is"); IO.write(olderThanMe != olderThanYou); }
                              // Java uses a C-like for loop
      for (year = yourAge; year <= myAge; year++)
        { IO.write("!"); }
      { IO.write(" - I am"); IO.write(myAge); }
    }

  } // demo
```

B8. In the examination kit you will find the files needed to build the pretty-printer you studied yesterday. You have also been given a complete listing of the `Parva.atg` grammar. Indicate the changes you would need to make to this grammar to build the required converter for the differences indicated:

(a) the keyword *boolean* and the *halt* statement [6 marks]

Changing `bool` *to* `boolean` *and* `halt` *to* `System.exit(0)` *is almost trivially easy*

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

      HaltStatement
***   =  "halt"                     (. CodeGen.append("System.exit(0)"); .)
        WEAK ";"                    (. CodeGen.append(";"); .) .
```

(b) the *for* statement [8 marks]

Modifying the for statement is easily done by inserting a simple conditional clause and a simple increment or decrement clause (and enclosing the controlling clauses in parentheses):

```
      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("; " + name + " <= "); .)
***        | "downto"               (. CodeGen.append("; " + name + " >= ");
                                       up = false; .)
         )
         Expression<out expType>    (. if (!assignable(des.type, expType))
                                         SemError("incompatible with control variable");
***                                    if (up) CodeGen.append("; " + name + "++)");
***                                    else CodeGen.append("; " + name + "--)"); .)
         Statement<indented>        (. loopLevel--; .)
      .
```

(c) *class* declarations [10 marks]

Enclosing the main function in a class declaration is straightforward, essentially requiring only a few more "appends", and the substitution of the classic main function signature:

```
      Parva
***   =                                  (. CodeGen.append("import Library.*;");
***                                         CodeGen.newLine();
***                                         CodeGen.newLine();
***                                         CodeGen.append("class " + srceName + " {");
***                                         CodeGen.newLine();
***                                         CodeGen.indentNewLine(); .)
         "void"                           (. Entry program = new Entry(); .)
         Ident<out program.name>
***      "(" ")"                          (. CodeGen.append("public static void main(String[] args)");
                                            program.kind = Entry.Fun;
                                            program.type = Entry.voidType;
                                            Table.insert(program);
                                            Table.openScope(); .)
         "{"                              (. CodeGen.append(" {"); CodeGen.indent(); .)
         { Statement<!indented> }
         WEAK "}"                         (. CodeGen.exdentNewLine(); CodeGen.append("}");
                                            if (debug) Table.printTable(OutFile.StdOut);
                                            Table.closeScope();
***                                         CodeGen.exdentNewLine(); CodeGen.newLine();
***                                         CodeGen.append("} // " + srceName ); .)
      .
```

(d) *const* declarations [16 marks]

Transforming constant declarations is a little more difficult. We need to be able to retrieve the type information that the Parva compiler deduces from the `const` *declaration, and make this explicit after the keyword* `final`. *It* `is` *easiest to place each declaration on its own line:*

```
      ConstDeclarations
***   = "const"
        OneConst
***     { WEAK ","                     (. CodeGen.newLine(); .)
          OneConst
***     } WEAK ";"
      .

      OneConst                         (. Entry constant = new Entry();
                                          ConstRec con; .)
      =  Ident<out constant.name>      (. constant.kind = Entry.Con; .)
***      "=" Constant<out con>         (. CodeGen.append("final ");
***                                       switch (con.type) {
***                                         case Entry.intType  :
***                                           CodeGen.append("int "); break;
***                                         case Entry.boolType :
***                                           CodeGen.append("boolean "); break;
***                                         case Entry.charType :
***                                           CodeGen.append("char "); break;
***                                         default :
***                                           CodeGen.append("int[] ");
***                                           SemError("invalid constant type"); break;
***                                       }
***                                       CodeGen.append(constant.name + " = " + con.name + ";");
                                          constant.type = con.type;
                                          Table.insert(constant); .) .
```

(e) the *write* statement [16 marks]

A `write` *statement in Parva may have multiple arguments, each of which must give rise to one call to the* `IO.write` *method. This is best handled by including each of these calls as statements within a { } block. Some students suggested that a statement like write(a, b) could be transformed into IO.write(a + b) using the oveloading of + that would work for a statement like write("hello", "knark") but not for one where a and b were integer expressions.*

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

```
          WriteElement
        }
***     ")"                          (. CodeGen.append("}"); .)
***     WEAK ";" .
```

Handling a `WriteElement` *is very simple:*

```
      WriteElement                   (. int expType;
                                        String str; .)
***   =                              (. CodeGen.append("IO.write("); .)
        (   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;
                                        } .)
***     )                            (. CodeGen.append("); "); .)
      .
```

(f) the *read* statement [16 marks]

A `read` *statement in Parva may have multiple arguments. As in the case of the* `write` *statement this is best handled by including each of the calls needed to methods of the* `IO` *library as statements within a { } block.*

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

A curious error that was found in many submissions was to generate an IO.write statement as the first component of all transformed read() statements. I suppose the authors had fallen into the trap of trying to generalize from the single example that had appeared in the question paper!

As the method call needed depends on the type of the designator, some care is needed to get the `ReadElement` *production correct:*

```
      ReadElement                    (. String str;
                                        DesType des; .)
***   = (   StringConst<out str>     (. CodeGen.append("IO.write(" + str + "); "); .)
        | Designator<out des>        (. if (des.entry.kind != Entry.Var)
                                          SemError("wrong kind of identifier");
                                        switch (des.type) {
                                          case Entry.intType:
***                                         CodeGen.append(" = IO.readInt(); "); break;
                                          case Entry.boolType:
***                                         CodeGen.append(" = IO.readBool(); "); break;
                                          case Entry.charType:
***                                         CodeGen.append(" = IO.readChar(); "); break;
                                          default:
                                            SemError("cannot read this type"); break;
                                        } .)
        )
      .
```

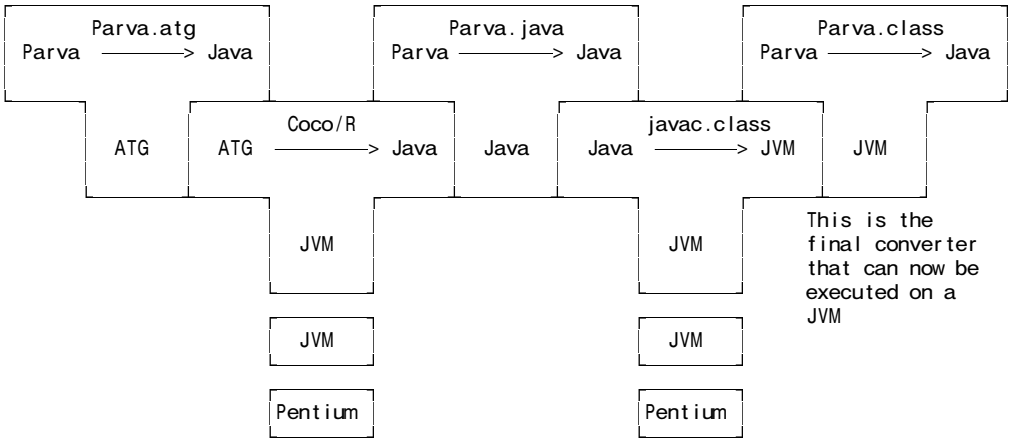
(Hint: You might be tempted to edit the files and produce a working version. It will suffice to indicate the changes on the grammar listing, or in your answer book. It is unlikely that you will be able to produce a complete working solution in the time available, but you should be able to give the examiner a very clear idea of what is required without necessarily getting every last action syntactically correct!)

B9. What is the name usually given to translators of this sort? [2 marks]

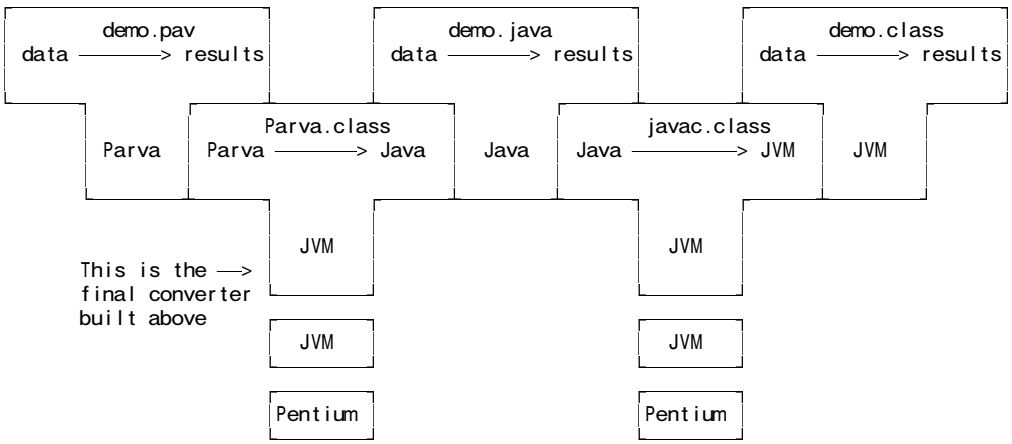
High-level compiler

B10. Draw two T diagrams - one illustrating how the translator is constructed and the second illustrating how a student's Parva program would actually be compiled and executed using this translator. Do this on the sheet provided, which is to be handed in with your answer book. [10 marks]

Building the translator



Executing a simple Parva program in the absence of a true Parva compiler



B11. You will surely have noticed that the pretty-printer strips comments from Parva programs.

(a) Why is this unimportant for the proposed use of the translator in the first year practical exam? [3 marks]

Essentially because nobody will "read" the Java code - all the development will be done in Parva. The Java compiler would ignore the comments anyway!

(b) Which facilities in Coco/R would allow you to retain the comments if required? (You do not have to give exact details of how this would be done.) [3 marks]

The comments could be defined as PRAGMAS and their text could be retrieved easily and appended to the output. It is actually quite tricky to get comments like this to look pretty, that is to put them back in exactly the right places. . Of course you would have to delete the existing COMMENTS section. I was pleased to see how many students had realised this. The previous day's effort had not been wasted, obviously!

Section C

(Summary of free information made available to the students 24 hours before the formal examination.)

A pretty-printer is a form of translator that takes source code and translates this into object code that is expressed

in the same language as the source. This probably does not sound very useful! However, the main aim is to format the "object code" neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

For example, a system that will read a set of EBNF productions, rather badly laid out as

```
Goal={One}.
One

= Two "plus" Four ".".

Four =Five {"is" Five|(Six Seven)}.
Five = Six [Six       ].
Six=  Two| Three    |"(*" Four "*)"|'(' Four ')'.
```

and produce the much neater output

```
Goal
  = { One } .

One
  = Two "plus" Four "." .

Four
  = Five { "is" Five | ( Six Seven ) } .

Five
  = Six  [ Six ] .

Six
  = Two | Three | "(*" Four "*)" | '(' Four ')' .
```

is easily developed by using a Cocol grammar *(supplied in the free information kit, but not reproduced here)* attributed with calls to a "code generator" consisting of some simple methods `append`, `indent`, `exdent` and so on.

As the first step in the examination candidates were invited to experiment with the EBNF system, and then to go on to develop a pretty-printer for programs expressed in a version of Parva, as described in the grammar below. A version of this grammar - spread out to make the addition of actions easy - was supplied in the examination kit, along with the necessary frame files.

Section D

(Summary of free information made available to the students 18 hours before the formal examination.)

Candidates were firstly informed that if one could assume that Parva programs submitted to it were correct, a basic pretty-printer for Parva programs could be developed from a fairly straightforward set of attributes added to the grammar seen in the free information below *(this attributed system was supplied in full in the auxiliary examination kit and is not listed).* These modifications assumed the existence of a `CodeGen` class - which essentially consisted of the same methods as students had seen embedded in the EBNF example supplied earlier in the day.

The attention of candidates was drawn to the way in which an `indented` argument to the `Statement` production was used to handle the indentation of the subsidiary statements found in *if, while* and *do-while* statements.

Candidate were informed that a better pretty-printer would incorporate some static semantic checking. in the spirit of that described in their textbook in chapter 12. They were presented with a rather longer grammar *(again, supplied in full in the auxiliary exam kit, and during the exam itself and is listed below),* derived from the system used in the final practical that not only would do pretty-printing, but also check for type mismatches, undeclared variables, break statements not embedded in loops and all those awful blunders that programmers sometimes make.

To prepare themselves to answer Section B of the examination they were encouraged to study the attributed grammar in depth, and warned that the questions in Section B would probe this understanding, and that they might be called on to make some modifications and extensions.

Free information

Context-free unattributed grammar for Parva

```
COMPILER Parva $NC
/* Parva level 1 grammar  - Coco/R for C# or Java */

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 ) "'" .

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

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva = "void" Ident "(" ")" "{" { Statement } "}" .
  Statement =   Block | ";" | Assignment | ConstDeclarations | VarDeclarations
              | IfStatement | WhileStatement | DoWhileStatement | ForStatement
              | BreakStatement | ContinueStatement | HaltStatement | ReturnStatement
              | ReadStatement | WriteStatement .
  Block = "{" { Statement } "}" .
  ConstDeclarations = "const" OneConst { "," OneConst } ";" .
  OneConst  = Ident "=" Constant .
  Constant = IntConst | CharConst | "true" | "false" | "null" .
  VarDeclarations = Type OneVar { "," OneVar } ";" .
  Type = BasicType [ "[]" ] .
  BasicType = "int" | "bool" | "char" .
  OneVar = Ident [ "=" Expression ] .
  Assignment = Designator ( "=" Expression | "++" | "--" ) ";" .
  Designator = Ident [  "[" Expression "]" ] .
  IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] .
  WhileStatement = "while" "(" Condition ")" Statement .
  DoWhileStatement = "do" Statement "while" "(" Condition ")" ";" .
  ForStatement = "for" Ident "=" Expression ( "to" | "downto" ) Expression Statement .
  BreakStatement = "break" ";" .
  ContinueStatement = "continue" ";" .
  HaltStatement = "halt" ";" .
  ReturnStatement = "return" ";" .
  ReadStatement = "read" "(" ReadElement { "," ReadElement } ")" ";" .
  ReadElement = ( StringConst | Designator ) .
  WriteStatement = "write" "(" WriteElement { "," WriteElement } ")" ";" .
  WriteElement = ( StringConst | Expression ) .
  Condition = Expression .
  Expression = AndExp { "||" AndExp } .
  AndExp = EqlExp { "&&" EqlExp } .
  EqlExp = RelExp { EqualOp RelExp } .
  RelExp = AddExp [ RelOp AddExp ] .
  AddExp = MultExp { AddOp MultExp } .
  MultExp = Factor { MulOp Factor } .
  Factor = Primary | "+" Factor | "-" Factor | "!" Factor .
  Primary =   Designator | Constant | "new" BasicType "[" Expression "]"
            | "(" ( "char" ")" Factor | "int" ")" Factor | Expression ")" ) .
  AddOp = "+" | "-" .
  MulOp = "*" | "/" | "%" .
  EqualOp = "==" | "!=" .
  RelOp = "<" | "<=" | ">" | ">=" .
  Ident = identifier .
  StringConst = stringLit .
  CharConst = charLit .
  IntConst = number .
END Parva.
```

Attributed Grammar for Parva pretty-printer with semantic checker

```
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.
```