RHODES UNIVERSITY

November Examinations - 2004

Computer Science 301 - Paper 2 - Solutions

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

Time 3 hours
Marks 180
Pages 7 (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 the full text of Section B. During the examination, candidates were given machine executable versions of the Coco/R compiler generator, access to a computer and machine readable copies of the questions.)

Section A [85 marks]

A1. Draw a diagram clearly depicting the various phases found in a typical compiler. Indicate which phases belong to the "front end" and which to the "back end" of the compiler. [10 marks]

This is pretty standard stuff, taken straight from the text:

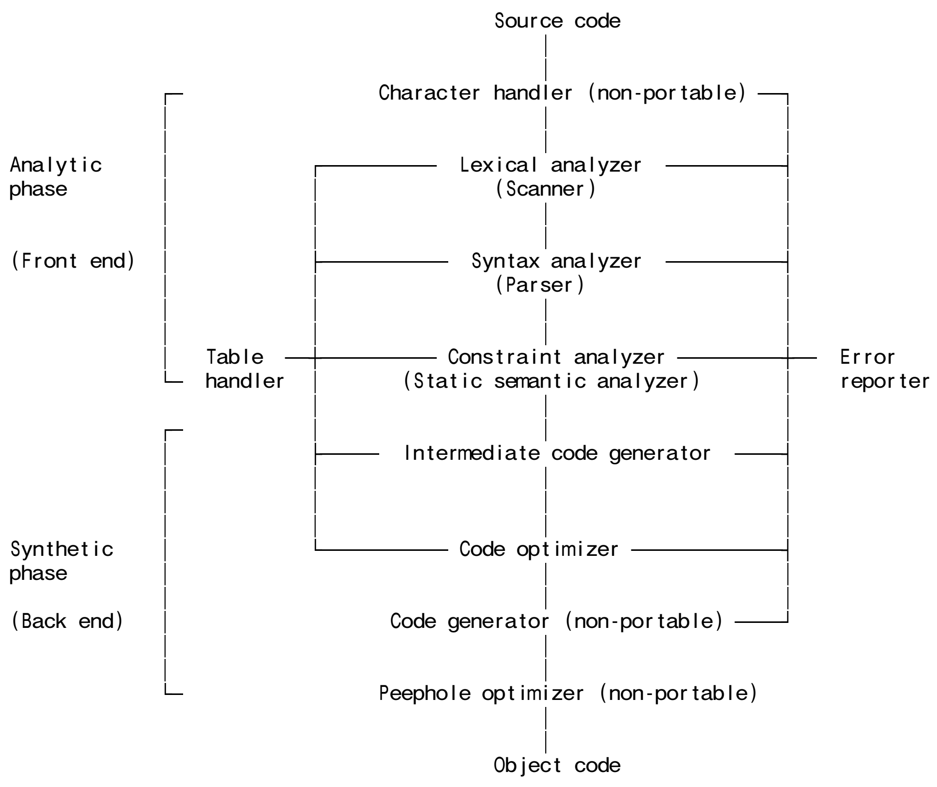

Source code
Character handler (non-portable)
Analytic phase
(Front end)
Lexical analyzer (Scanner)
Syntax analyzer (Parser)
Table handler
Constraint analyzer (Static semantic analyzer)
Error reporter
Intermediate code generator
Synthetic phase
(Back end)
Code optimizer
Code generator (non-portable)
Peephole optimizer (non-portable)
Object code

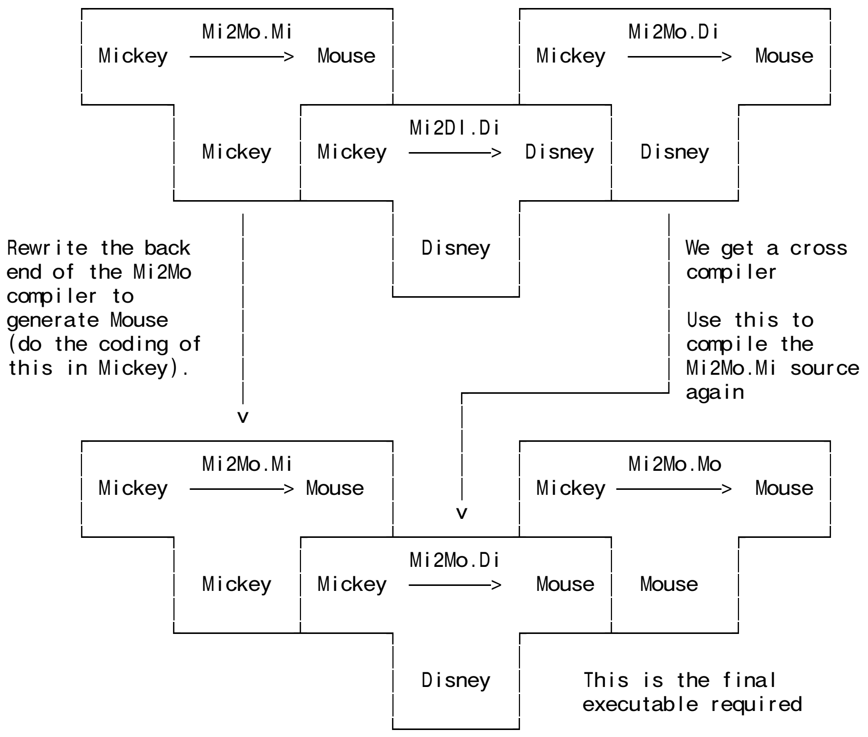
Figure 2.5 Structure and phases of a compiler

A2. (a) What is meant by the term "self-compiling compiler"? [3 marks]

A compiler hosted in the language it is intended to compile, so that it can compile (regenerate) itself.

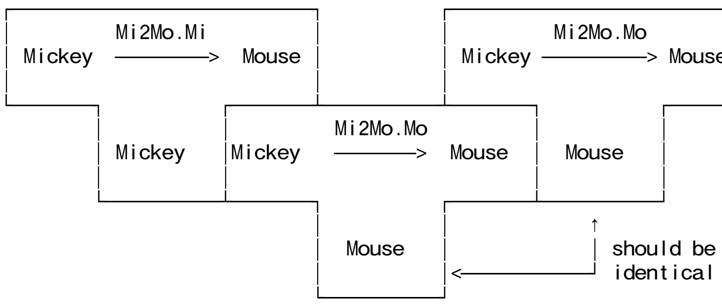
(b) Describe (with the aid of T-diagrams) how you would perform a "half bootstrap" of a compiler for language Mickey, given that you have access to the source and object versions of a compiler for Mickey that can be executed on machine Disney, and wish to produce a self-compiling Mickey-Mouse compiler for language Mickey that can be executed on machine Mouse. [8 marks]

This is a variation on one of a number of such bootstrapping examples discussed in the course:



(c) Self-compiling compilers have to satisfy a self-consistency test. Explain this in terms of a suitably annotated T-diagram. [3 marks]

Again, this is very straightforward:



(A set of T-diagrams appears on an appendix to this paper. You may tear this off, complete it, and hand it in with your answer book.)

A3. The following Cocol grammar may be familiar. It describes a set of EBNF productions that can incorporate Wirth's metabrackets { } [and].

```
COMPILER EBNF $CN
/* Parse a set of EBNF productions
   P.D. Terry, Rhodes University, 2004 */

CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline   = "_" .
  digit     = "0123456789" .
  noquote1  = ANY - "'" .
  noquote2  = ANY - '"' .

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

COMMENTS FROM "(*" TO "*)"  NESTED

IGNORE  CHR(9) .. CHR(13)

PRODUCTIONS
```

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

(a) Derive the `PRODUCTIONS` section for an equivalent grammar that describes, but does not use, the metabrackets [ ] { and }. [4 marks]

This is a straightforward conversion of the sort often demonstrated in the course:

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

There are other grammars that seem to do the trick as well, for example:

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

but you might like to consider whether these two are really equivalent.

(b) Show how the productions would be attributed so as to parse a set of productions given in the Wirth notation and reproduce them one to a line in the alternative EBNF notation that uses Kleene closure symbol * and ε. For example, a production like

```
Program = [ Header ] { Statement } .
```

should be transformed to

```
Program = ( Header | ε ) ( Statement )* .
```

For convenience, the grammar above has been spread out on an appendix to this paper. [12 marks]

A complete Java version of the solution is given below. Of course only the PRODUCTIONS section was required in the examination.

```
import Library.*;

COMPILER EBNF $CN
/* Parse a set of EBNF productions and convert to one form of EBNF
   P.D. Terry, Rhodes University, 2004 */

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  digit    = "0123456789" .
  noquote1 = ANY - "'" .
  noquote2 = ANY - '"' .
```

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

COMMENTS FROM "(*" TO "*)"  NESTED

IGNORE  CHR(9) .. CHR(13)

PRODUCTIONS
   EBNF
   = { Production } EOF .

   Production
   = nonterminal           (. IO.write(token.val + " "); .)
   "="                     (. IO.write("= "); .)
   Expression
   "."                     (. IO.writeLine(". "); .)
   .

   Expression
   = Term
   { "|"                   (. IO.write("| "); .)
   Term } .

   Term
   = Factor { Factor } .

   Factor
   =   nonterminal         (. IO.write(token.val + " "); .)
     | terminal            (. IO.write(token.val + " "); .)
     | "["                 (. IO.write("( "); .)
       Expression "]"      (. IO.write(" | eps ) "); .)
     | "("                 (. IO.write("( "); .)
       Expression
       ")"                 (. IO.write(") "); .)
     | "{"                 (. IO.write("( "); .)
       Expression
       "}"                 (. IO.write(")* "); .)
     .
END EBNF.
```

A4. Chomsky classified grammars into four types, sometimes known as types 0, 1, 2, 3. The classification depended on the form of the productions. Computer Scientists tend to use more descriptive names for these types - for example a type 0 grammar is often called "unrestricted".

(a) What are the names commonly used to describe the other types? Your answer should take the form

A Type 1 grammar is also called ...
A Type 2 grammar is also called ...
A Type 3 grammar is also called ...

[3 marks]

Solution:

A Type 1 grammar is also called Context-sensitive
A Type 2 grammar is also called Context-free
A Type 3 grammar is also called Regular

(b) Consider a simple grammar with two non-terminals { A, B }, four terminals { a, b, c, d } and productions defined in EBNF as follows:

```
A = "a" B | B "b" B .
B = "c" A | "d" .
```

Which type of grammar is exemplified by these productions? [1 mark]

This is most tightly defined as "context-free" or "type 2"

(c) Is this a reduced grammar? Explain your answer. [2 marks]

Yes it is. All non-terminals can derive strings containing only terminals, and all non-terminals can be reached in a derivation starting from the goal symbol (regardless of whether this is thought to be A or B).

(d) Suggest simple alterations to the production set that would cause the grammar to be classified as being of each of the other forms (these grammars do not have to represent any meaningful language). Your answers should take the form

A Type X grammar would result if we had productions like ...

[3 marks]

There are many possible solutions. Among these might be:

A Type 3 grammar would result if we had productions like ...

```
A   = "a" B | "b" B .
B   = "c" A | "d" .
```

A Type 1 grammar would result if we had productions like ...

```
A   = "a" B | B "b" B .
A B = "c" A | "d" .
```

A Type 0 grammar would result if we had productions like ...

```
A   = "a" B | B "b" B .
"d" = "c" A | "d" .
```

A5. The following grammar attempts to describe expressions incorporating addition, subtraction, multiplication, division and exponentiation, with the correct precedence and associativity of the operators.

```
Expression  = Term { ( "+" | "-" ) Term } .
Term        = Factor { ( "*" | "/" ) Factor } .
Factor      = Primary [ "↑" Expression ] .
Primary     = "a" | "b" | "c" | "(" Expression ")" .
```

(a) Is it an LL(1) grammar? If not, why not, and can you find a suitable grammar that *is* LL(1)? [8 marks]

*Solution: It is not an LL(1) grammar. It might be difficult to see this at first, or it may be easy. It's expressions like a↑b*c that give things away - the grammar is ambiguous in that regard.*

A manual analysis would go something like this. Rewrite the productions in BNF form:

```
Expression  = Term MoreTerms .
MoreTerms   = ( "+" | "-" ) Term MoreTerms | ε .
Term        = Factor MoreFactors .
MoreFactors = ( "*" | "/* ) Factor MoreFactors | ε .
Factor      = Primary Exponent .
Exponent    = "↑" Expression | ε .
Primary     = "a" | "b" | "c" | "(" Expression ")" .
```

Clearly "Rule 1" is satisfied. Any problems would come about if "Rule 2" were violated.
To check Rule 2 we need to examine the intersections of

```
FIRST(MoreTerms)    = { "+" , "-" }
FOLLOW(MoreTerms)   = { EOF  "+"  "-"  "*"  "/"  ")" }

FIRST(MoreFactors)  = { "*" , "/" }
FOLLOW(MoreFactors) = { EOF  "+"  "-"  "*"  "/"  ")" }

FIRST(Exponent)     = { "↑" }
FOLLOW(Exponent)    = { EOF  "+"  "-"  "*"  "/"  ")" }
```

Computing these FOLLOW sets is a bit tedious. For example

```
FOLLOW(Exponent)    = FOLLOW(Factor)
                    = FIRST(MoreFactors) U FOLLOW(Term)
```

```
                                 = FIRST(MoreFactors) U FIRST(MoreTerms) U FOLLOW(Expression)
                                 = { EOF  "+"  "-"  "*"  "/"  ")" }
```

and similarly for the others. The classic solution to this dilemma is

```
Expression   =  Term { ( "+" | "-" ) Term } .
Term         =  Factor { ( "*" | "/" ) Factor } .
Factor       =  Primary [ "↑" Factor ] .
Primary      =  "a" | "b" | "c" | "(" Expression ")" .
```

The easy way to do the calculation of the FIRST and FOLLOW sets is to submit the following to Coco, which is an acceptable exam solution for those who realize this can be done

```
COMPILER Expression $TF /* pragmas to test the grammar properly */

PRODUCTIONS
  Expression  = Term MoreTerms .
  MoreTerms   = ( "+" | "-" ) Term MoreTerms |  .
  Term        = Factor MoreFactors .
  MoreFactors = ( "*" | "/" ) Factor MoreFactors |  .
  Factor      = Primary Exponent .
  Exponent    = "↑" Expression |  .
  Primary     = "a" | "b" | "c" | "(" Expression ")" .
END Expression.
```

(b) Assume that you have available a suitable scanner method called `getSym` that can recognize the terminals of this language and map them appropriately to the members of the following enumeration

```
EOFSym, noSym, aSym, bSym, cSym, addSym, subSym, mulSym, divSym, expSym, lParSym, rParSym
```

Develop a hand-crafted recursive descent parser for recognizing valid expressions. Your parser can take drastic action if an invalid expression is detected - simply produce an appropriate error message and then terminate parsing.

(You are not required to write any code to implement the `getSym` method, and you can ignore any complications that might arise if the defining grammar is non-LL(1).) [12 marks]

Solution would be on the following lines:

```
static void Expression () {
  Term();
  while (Sym.kind == addSym || sym.kind == subSym) {
    getSym();
    Term();
  }
}

static void Term () {
  Factor();
  while (Sym.kind == mulSym || sym.kind == divSym) {
    getSym();
    Factor();
  }
}

static void Factor () {
  Primary();
  if (sym.kind == expSym) {
    getSym();
    Expression();
  }
}

static void Primary () {
  switch (sym.kind) {
    case aSym:
    case bSym:
    case cSym:
      getSym(); break;
    case lParSym;
      getSym(); Expression();
      accept(rParSym, ") expected");
```

```
          break;
        default:
          abort("invalid primary");
          break;
      }
    }
```

A6. (a) *Scope* and *Existence/Extent* are two terms that come up in any discussion of the implementation of block-structured languages. Briefly explain what these terms mean, and the difference between them. [6 marks]

Scope is a compile-time concept - essentially it refers to the "area" of code in which an identifier can be recognized. An extract from the text is relevant here, though all this information was not required for a full solution.

Languages like Pascal, C#, Java - and even Parva - are said to be **block-structured**. The concept of **scope** should be familiar to readers experienced in developing code in block-structured languages, although it causes confusion to some beginners. In such languages, the "visibility" or "accessibility" of an identifier declared in a *Block* is limited to that block, and to blocks themselves nested within that block. Some rule has to be applied when an identifier declared in one block is *redeclared* in one or more nested blocks. This rule differs from language to language, but in many cases such redeclaration is allowed, on the understanding that it is the innermost accessible declaration that will apply to any particular use of that identifier.

Existence or Extent is a run-time concept - essentially it refers to the "real time" during which storage need be allocated to a data structure of whatever complexity. Another extract from the text is relevant here:

Just as the stack concept turns out to be useful for dealing with the compile-time accessibility aspects of *scope* in block-structured languages, so too do stack structures provide a solution for dealing with the run-time aspects of *extent* or *existence*. Each time a function is called, it acquires a region of free store for its local variables and arguments - an area which can later be freed when control returns to the caller. On a stack machine this becomes almost trivially easy to arrange, although it may be more obtuse on other architectures. Since function activations strictly obey a first-in-last-out scheme, the areas needed for their local working store can be carved out of a single large stack. Such areas are usually called **activation records** or **stack frames**.

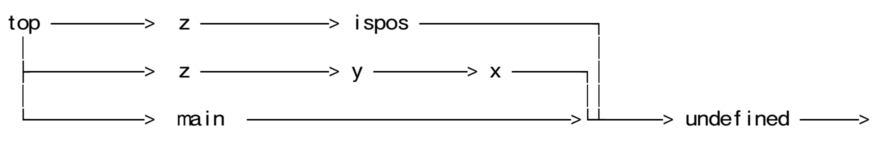
(b) Scope rules for a simple block-structured language like Parva can be implemented by making use of a suitable data structure for the symbol table. Show what such a structure might look like when a top-down one-pass compiler reached each of the points marked (1) and (2), if it compiled the program below. [10 marks]

```
void main () {
  int x = 10, y, z;
  while (x > 0) {
    read(y);
    bool isPosY = y > 0;
    int z = x / 2;
    x = x - y;
    // point (1)
  }
  int a = x + y + z;
  // point (2)
}
```

Solution:

Various solutions are possible, but they all rely to some extent on a stack or farm of stacks. The sort of structure exemplified in our case studies was:

At point 1 we might have



At point 1 we might have

```
top ─────> a ────> z ────> y ──────> x ─────┐
 │                                          │
 └─────────> main ──────────────────────────>└─────> undefined ────>
```

Section B [95 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.

2004 has been a great year for anniversaries: 10 years of democracy; 10 years of 24-hour compiler course examinations; 100 years of excellence at Rhodes University; 60 years since the allies invaded Europe on D-Day; 40 years since the Beatles invaded the United States; 50 years of MacDonald's Hamburgers; 50 years of Elvis Presley recordings. The list is endless.

It is also 50 years after Backus started work on the programming language FORTRAN.

Regular readers of this column - the Compiler Course Examination Archives (CCEA) - will know that this time of the year usually sees a crisis develop in the Computer Science Department, and this year is no exception. As part of the Rhodes Centenary Celebrations, each department has been mounting exhibitions that incorporate their most important relics of the past. A potential rich research funder is to visit our exhibition on the day after tomorrow, and a lot is at stake. His silver hair suggests that he only ever programmed in FORTRAN, and so it is slightly unfortunate that we have not found a FORTRAN compiler, let alone one that targets the ground-breaking PVM (Parva Virtual Machine) on which the Department's research reputation is increasingly based.

"No problem", exclaimed the illustrious Head of Department. "Write one! I know that the first FORTRAN compiler is reputed to have taken 18 person-years of effort, but we don't need a full FORTRAN compiler - we need only demonstrate a carefully chosen subset compiler and that should easily convince the potential funder that we have the Real Thing".

Very simple FORTRAN programs are not that hard to code or understand. They have a single program unit that starts with a `PROGRAM` line and ends with an `END` line. In between these come, firstly, a list of variable declarations, and then, secondly a sequence of executable statements. In the original FORTRAN, only upper-case characters were allowed, but today it is generally taken as a case-insensitive language. Only one statement may appear on a line, so a simple example that would impress our visitor immensely might be provided by:

```
    PROGRAM Greeting
C:  Comments start with C: and go on to the end of the line
    INTEGER Year, Born
    Year = 2004
    PRINT *, 'When were you born?'
    READ  *, Born
    PRINT *, 'That means you''re ', Year - Born, ' years old!', EOLN
    STOP
    END
```

The asterisks in the `READ` and `PRINT` statements denote input from the "standard input" (keyboard) and output to the "standard output" (screen) devices respectively. The asterisk in `READ` statements is followed by a list of designators in a familiar way, and in `PRINT` statements by an obvious list of expressions and strings. Unlike Java, FORTRAN literal strings are bracketed with single apostrophes. If a string is to contain an apostrophe, this is denoted by two apostrophes in succession, as in the example just given, which would display something like:

```
That means you're 59 years old!
```

if the program were executed. Other escape sequences like the familiar `\n` and `\t` found in Java strings are not allowed. Although it is not really part of standard FORTRAN, we suggest using the token `EOLN` to represent "output an end of line sequence".

For the purposes of this exercise, limit variables to being of only two types, denoted by `INTEGER` (int) and `LOGICAL` (Boolean), and demand that they be declared as in the following examples:

```
      INTEGER I, J, K, List(12)
      LOGICAL Sieve(4000), IsEasy, IsOld, CanRetire
      INTEGER N, Age
```

where arrays are indicated (and storage automatically allocated to them) by indicating the uppermost permitted value of the (integer) subscript in parentheses, for those identifiers that are to denote arrays.

Arithmetic (integer) expressions can contain the usual `+`, `-`, `*` and `/` operators. In forming logical (Boolean) expressions the tokens `.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.`, `.GE.`, `.AND.`, `.OR.` and `.NOT.` are used, and the logical constants are denoted by `.FALSE.` and `.TRUE.` Within expressions, array elements are selected using index expressions contained in `(` round `)` parentheses rather than `[` square `]` brackets. All this rather clumsy notation comes about because of the limited character sets available on computers 50 years ago. Precedence rules are effectively the same as we still have in Java. Here are some examples of simple assignments in FORTRAN:

```
      IsOld = Age .GE. 25
      Profit = Items * (Sell - Cost)
      CanRetire = Age .GT. 55 .AND. Pension .GT. 100000 .OR. WifeInsists
      Average = (List(1) + List(2) + List(3)) / 3
```

Where FORTRAN differs significantly from the languages most familiar to you is in the way in which it handles branching and looping. As FORTRAN evolved, so too did its control structures, but our old visitor might not recognize all of those, so we should rather cater for the traditional forms. Chief among these is the `GOTO` statement. An executable FORTRAN statement can be associated with a unique label, and such labelled statements can be the target of `GOTO` statements, as exemplified by the mindless program:

```
      PROGRAM Parrot
   10 PRINT *, 'Pretty Polly '
      GOTO 10
      END
```

Of course, one needs somewhat more sophistication. A rather strange statement found in the original FORTRAN is the so-called "arithmetic `IF`" statement, exemplified by:

```
      IF (A - B * C) 10, 20, 30
```

The dynamic semantics of this statement form are as follows: the parenthesized expression - which has to be "arithmetic" rather than "logical" - is evaluated, followed by one of a `GOTO 10` (the first label) if the result is negative, a `GOTO 20` (the second label) if the result is zero, and a `GOTO 30` if the result is positive. All three labels have to be provided (and, of course, each label has to be attached to a statement somewhere within the program). Here is a more complete example:

```
      PROGRAM Decide
      INTEGER I, J
   90 READ *, I, J
      IF (I - J) 20, 500, 500
   20 PRINT *, 'I is less than J'
      GOTO 30
  500 PRINT *, 'I is greater than or equal to J'
   30 STOP
      END
```

This may strike you as a bit tortuous, and it is not hard to see that a program with many `GOTO` statements and labels (which could be assigned to statements in any order) can become hard for a human reader to understand.

A little later in the history of FORTRAN came the introduction of the "logical `IF`" statement. In this statement the parenthesized expression after `IF` has to be "logical" rather than "arithmetic", and is followed by a single statement which is executed if the expression evaluates to *true*. Again some examples will clarify:

```
      IF (A .GT. B) PRINT *, 'A is greater than B'

      Total = 0
   10 READ *, I
      Total = Total + I
      IF (I .NE. 0) GOTO 10
      PRINT *, 'Total = ', Total
```

This "logical `IF`" statement did not provide for an `ELSE` clause (that came even later in the history of

FORTRAN) and the auxiliary statement could only be one of a limited set of possibilities - it could be a `READ`, `PRINT`, `STOP`, `CONTINUE`, `GOTO`, or an assignment, but not another `IF` statement.

The `STOP` statement does the obvious thing (halts program execution) and the `CONTINUE` statement does "nothing" - it is a useful way of introducing an extra label into a program if that is ever needed.

The last control statement we should like to demonstrate to our visitor is the `WHILE` statement, which is exemplified by the following code (which also incorporates simple array handling):

```
      Total = 0
      N = 1
      Read *, Item
      WHILE (Item .NE. 0)
        List(N) = Item
        N = N + 1
        READ *, Item
      ENDWHILE
```

Here the parenthesized expression in the `WHILE` statement must be a "logical expression", and the body of the loop consists of the statements between the `WHILE` statement itself and the distinctive `ENDWHILE` statement. `WHILE` loops can be nested, and `ENDWHILE` statements can be labelled, so a larger example might be

```
      I = 0
      WHILE (I .LE. 10)
        J = 0
        WHILE (J .LE. 0)
          PRINT *, I * J
        ENDWHILE
        PRINT *, EOLN
  10  ENDWHILE
```

`WHILE` and `ENDWHILE` statements cannot form part of a "logical `IF`" statement.

Save the honour of the Department! Spend the next 24 hours using Coco/R to develop a subset FORTRAN compiler that targets the PVM and handles the set of statements loosely described above, and then present a report and a Cocol grammar showing how you would do this. To assist you in this task we shall provide you with an attributed grammar and the usual support modules, from which a working Parva compiler/interpreter system can be constructed. This is essentially the same as the one which you explored in the practical course, but with the part of the compiler that deals with expressions already modified to incorporate the C#/Java rules for precedence. It should be apparent that large parts of the Parva compiler can be incorporated into the FORTRAN compiler almost unchanged, and you are encouraged to do so, or to modify components (such as the PVM or symbol table handlers) as you see fit. The Parva system forms part of a kit that also includes various other sample FORTRAN programs that you may find useful in developing and testing your compiler.

Solution:

What follows is a commentary on those parts of the Parva system that could be hacked to get the desired result within the limits of the exercise. There is quite a lot to the exercise as a whole. Parts of the solution are very easily implemented. For example, the strange operators used in logical expressions lead to productions like those below, and once you have seen one of these you should see how to do them all!

```
  Constant<out ConstRec con>          (. con = new ConstRec(); .)
  =   IntConst<out con.value>         (. con.type = Entry.intType; .)
    | ".TRUE."                        (. con.type = Entry.boolType; con.value = 1; .)
    | ".FALSE."                       (. con.type = Entry.boolType; con.value = 0; .)
  .

  RelOp<out int op>
  =                                   (. op = CodeGen.nop; .)
   (    ".LT."                        (. op = CodeGen.clt; .)
      | ".LE."                        (. op = CodeGen.cle; .)
      | ".GT."                        (. op = CodeGen.cgt; .)
      | ".GE."                        (. op = CodeGen.cge; .)
   ) .

  EqualOp<out int op>                 (. op = CodeGen.nop; .)
  =     ".EQ."                        (. op = CodeGen.ceq; .)
      | ".NE."                        (. op = CodeGen.cne; .)
  .
```

Two other examples were buried in the parsing hierarchy for *Expressions*:

```
Expression<out int type>           (. int type2;
                                      Label shortcircuit = new Label(!known); .)
=  AndExp<out type>
   { ".OR."                        (. CodeGen.booleanOp(shortcircuit, CodeGen.or); .)
     AndExp<out type2>             (. if (!isBool(type) || !isBool(type2))
                                        SemError("logical operands needed");
                                      type = Entry.boolType; .)
   }                               (. shortcircuit.here(); .)
.

AndExp<out int type>               (. int type2;
                                      Label shortcircuit = new Label(!known); .)
=  EqlExp<out type>
   { ".AND."                       (. CodeGen.booleanOp(shortcircuit, CodeGen.and); .)
     EqlExp<out type2>             (. if (!isBool(type) || !isBool(type2))
                                        SemError("logical operands needed");
                                      type = Entry.boolType; .)
   }                               (. shortcircuit.here(); .) .
```

Most of this hierarchy survived intact, but strictly the *Primary* production has to be simplified

```
Primary<out int type>              (. type = Entry.noType;
                                      DesType des;
                                      ConstRec con; .)
=    Designator<out des>           (. type = des.type;
                                      switch (des.entry.kind) {
                                        case Entry.Var:
                                          CodeGen.dereference();
                                          break;
                                        default:
                                          SemError("wrong kind of identifier");
                                          break;
                                      } .)
   | Constant<out con>             (. type = con.type;
                                      CodeGen.loadConstant(con.value); .)
   | "(" Expression<out type> ")"
.
```

The original system had an `unescape` method for handling escape sequences like \n and \t. The modified system needs a somewhat simpler, though similar implementation:

```
static String unescape(String s) {
/* Replaces '' escape sequences in s by single quotes ' */
  StringBuffer buf = new StringBuffer();
  int i = 0;
  while (i < s.length()) {
    if (s.charAt(i) == '\'') i++;
    buf.append(s.charAt(i)); i++;
  }
  return buf.toString();
}
```

This has to be seen in conjunction for a revised token definition for a `stringLit`:

```
CHARACTERS
  stringCh   = ANY - "'" - control .
TOKENS
  stringLit  = "'" { stringCh | "''" } "'" .
```

To make the system insensitive to variations in case requires the addition of a simple directive and a modification to the *Ident* production, and to make the end of line significant requires the introduction of an `eol` token

```
IGNORECASE
CHARACTERS
  lf         = CHR(10) .
TOKENS
  eol        = lf .

 ...

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

Handling comments is easily achieved with the directive as below. People familiar with FORTRAN will see this as an (acceptable) kludge!

```
COMMENTS FROM "C:" TO lf
```

So far as the `PRODUCTIONS` go, the equivalent of the "`void main`" function parser requires fairly minimal changes to some key words, and the separation of declarations and executable statements:

```
ToyFort
=  { eol } "PROGRAM"              (. Entry program = new Entry();
                                     boolean endWhile; .)
   Ident<out program.name> EOLS   (. program.kind = Entry.Fun;
                                     program.type = Entry.voidType;
                                     Table.insert(program);
                                     StackFrame frame = new StackFrame();
                                     Table.openScope();
                                     Label DSPLabel = new Label(known);
                                     CodeGen.openStackFrame(); .)
   { VarDeclarations<frame> }
   { LabelledStatement<out endWhile> }
   "END" EOLS                    (. if (debug) Table.printTable(OutFile.StdOut);
                                     CodeGen.fixDSP(DSPLabel.address(), frame.size);
                                     CodeGen.checkStop();
                                     LabelTable.checkLabels();
                                     Table.closeScope();
                                     if (loopCount > 0)
                                       SemError("unterminated loops"); .) .
```

However, there are several subtleties here:

- Note the call to `CodeGen.checkStop` which generates the `PVM.trap` opcode suggested in one of the examples supplied as part of the kit.
- The test on `loopCount` comes about because of the way in which *WhileStatement*s are handled. Each of these requires an `ENDWHILE` statement, but of course those might have been omitted in error.
- The call to `LabelTable.checkLabels` is needed to make sure that the targets of all GOTO and arithmetic IF statements have been properly defined. This should have been familiar from the last practical in the course which had implemented `goto` statements in Parva.
- The productions relating to `ConstDeclarations` fall away completely.

It is convenient, though not obligatory, to use the mandatory `eol` token with which many statements end as a synchronization point:

```
EOLS = SYNC eol { eol } .
```

Note that this production also allows for completely blank lines to follow statements.

Variable declarations are virtually identical to those in the Parva compiler:

```
VarDeclarations<StackFrame frame>     (. int type = Entry.intType; .)
=  (   "INTEGER"
     | "LOGICAL"                      (. type = Entry.boolType; .)
   )
   OneVar<frame, type>
   { WEAK "," OneVar<frame, type> }
   EOLS .
```

However, the production for *OneVar* needs tweaking, so as to auto-generate the code needed to reserve array space on the he

```
OneVar<StackFrame frame, int type>    (. int size; .)
=                                     (. Entry var = new Entry(); .)
   Ident<out var.name>                (. var.kind = Entry.Var;
                                         var.type = type;
                                         var.offset = frame.size;
                                         frame.size++; .)
   [                                  (. var.type++; .)
     "(" IntConst<out size> ")"       (. CodeGen.loadAddress(var);         //+++++++++
                                         CodeGen.loadConstant(size + 1);  //+++++++++
                                         CodeGen.allocate();              //+++++++++
                                         CodeGen.assign(var.type);        //+++++++++ .)
   ]                                    (. Table.insert(var); .) .
```

Note how the size of the array is one larger than the number specified in the declaration. In fact FORTRAN arrays were indexed `1 ... N` and not `0 ... N-1`. But this complication was not mentioned in the problem and you could not have been expected to know that.

On now to the ways of handling executable statements. The quirks of the "logical if" mean that we have to break statements down into two categories. Any standalone executable statement can be labelled, but *IfStatement*, *WhileStatement* and `EndWhileStatement` are not permitted within a "logical IF" statement:

```
LabelledStatement<out boolean endWhile>
/* These are not permitted within a logical IF statement */
= [ Label ]                          (. endWhile = false; .)
  (   Statement
    | IfStatement
    | WhileStatement
    | EndWhileStatement              (. endWhile = true; .)
  ) .
```

In Fortran a "blank" statement could not be labelled, so far as I recall (one used `CONTINUE` to overcome that). Nor could several labels be attached to the same statement. So the obvious temptation to allow a standalone label to be a "statement", while it appears to work for this system, is technically incorrect!

Notice the way of recognizing an `ENDWHILE` statement as something special so that the *WhileStatement* parser can react accordingly (see later discussion).

The simpler statements follow as:

```
Statement
/* these are all permitted as the subsidiary within a logical IF statement */
=  SYNC (   Assignment
          | GoToStatement
          | StopStatement
          | ContinueStatement
          | ReadStatement
          | PrintStatement
        ) .
```

Of these, some differ only marginally from Parva and are easily handled:

```
StopStatement
= "STOP" EOLS                        (. CodeGen.leaveProgram(); .) .

ContinueStatement
= "CONTINUE" EOLS .

ReadStatement
= "READ" "*" { WEAK "," ReadElement } EOLS .

ReadElement                          (. DesType des; .)
= Designator<out des>                (. if (des.entry.kind != Entry.Var)
                                          SemError("wrong kind of identifier");
                                        switch (des.type) {
                                        case Entry.intType:
                                        case Entry.boolType:
                                          CodeGen.read(des.type); break;
                                        default:
                                          SemError("cannot read this type"); break;
                                       } .) .

PrintStatement
= "PRINT" "*" { WEAK "," PrintElement } EOLS .

PrintElement                         (. int expType;
                                        string str; .)
=   StringConst<out str>             (. CodeGen.writeString(str); .)
  | "EOLN"                           (. CodeGen.writeLine(); .)
  | Expression<out expType>          (. switch (expType) {
                                          case Entry.intType:
                                          case Entry.boolType:
                                            CodeGen.write(expType); break;
                                          default:
                                            SemError("cannot write this type"); break;
                                        } .) .
```

The other statements are more fun. The simple *GoToStatement* uses ideas that students should have seen from an earlier practical, though a little massaging turns out to be advantageous:

```
Label                              (. int label; .)
= IntConst<out label>              (. LabEntry labelEntry = LabelTable.find(label);
                                      if (labelEntry == null)
                                        LabelTable.insert(new LabEntry(label, new Label(known)));
                                      else if (labelEntry.label.isDefined())
                                        SemError("label " + label + " already defined");
                                      else labelEntry.label.here(); .) .

GoToStatement                      (. Label lab; .)
= "GOTO" Target<out lab> EOLS      (. CodeGen.branch(lab); .) .

Target<out Label lab>              (. int target; .)
= IntConst<out target>             (. LabEntry labelEntry = LabelTable.find(target);
                                      if (labelEntry == null) {
                                        lab = new Label(!known);
                                        LabelTable.insert(new LabEntry(target, lab));
                                      }
                                      else lab = labelEntry.label; .) .
```

The harder parts of this question relate to the *IfStatement* and to the *WhileStatement*, and these require insight into the system at a deeper level than the points discussed above. Both forms of `IF` statement start with the keyword `IF`, and the parser has to be driven "semantically" by checking the type of the controlling expression. Once one has realized and appreciated this, the rest is not too bad. The "logical IF" code generation closely matches that in Parva. The "arithmetic IF" is probably best handled by extending the PVM to have an `arif` instruction that (uniquely) takes three arguments, and which reacts to the arithmetic value of the controlling *Expression* that would have found its way to the top of the runtime stack just before the `arif` instruction is encountered:

The interpretation of this could be on the lines of

```
case PVM.arif:
  tos = pop();
  if (tos < 0) cpu.pc = mem[cpu.pc];
  else if (tos == 0) cpu.pc = mem[cpu.pc + 1];
  else cpu.pc = mem[cpu.pc + 2];
  break;
```

This necessitates various other cosmetic changes to the PVM interpreter class that need not be discussed here (for example, the `ARIF` string would need to be associated with a numeric value for `PVM.arif`. All of this should have been familiar to students who had completed practical exercises involving the PVM).

The *IfStatement* parser becomes

```
IfStatement                        (. int type;
                                      Label less, equal, greater;
                                      Label falseLabel = new Label(!known); .)
=  "IF" "(" Expression<out type> ")"
   (                               (. if (isBool(type))
                                        SemError("integer expression needed"); .)
        Target<out less> ","
        Target<out equal> ","
        Target<out greater>        (. CodeGen.choose(less, equal, greater); .)
        EOLS
     |                             (. if (!isBool(type))
                                        SemError("logical expression needed"); .)
                                   (. CodeGen.branchFalse(falseLabel); .)
        Statement                  (. falseLabel.here(); .)
   ) .
```

where reference has been made to a new code generating routine whose implementation is very straightforward:

```
public static void choose(Label less, Label equal, Label greater) {
// Generates selector for arithmetic if statement
  emit(PVM.arif);
  emit(less.address());
  emit(equal.address());
  emit(greater.address());
```

```
      }
```

I suspect that the implementation of a *WhileStatement* is actually trickier than most people realized at first, though the ultimate solution proposed here is almost delightfully simple (like most of the extensions to a Parva like language!). What is required is that an arbitrary sequence of statements must be allowed to follow the *WhileStatement* itself - this sequence can of course include further *WhileStatements*. A production like

```
WhileStatement = "WHILE" "(" Expression ")" EOLS { LabelledStatement } .
```

would (and does) lead to a non-LL(1) system. Normally these are pretty useless, but we have the option of breaking out of the loop from within the associated actions, thus:

```
WhileStatement                       (. int type;
                                        boolean endWhile;
                                        Label startLoop = new Label(known); .)
=  "WHILE" "(" Expression<out type>  (. if (!isBool(type))
                                          SemError("logical expression needed"); .)
   ")"  EOLS                         (. Label loopExit = new Label(!known);
                                        CodeGen.branchFalse(loopExit);
                                        loopCount++; .)
   { LabelledStatement<out endWhile> (. if (endWhile) {
                                          loopCount--;
                                          break;
                                        } .)
   }                                 (. CodeGen.branch(startLoop);
                                        loopExit.here(); .) .
```

The code generation itself is trivially easy - it is just as it is for the `while` loop in Parva.

Two other subtleties are that `ENDWHILE` can only be allowed "within" a `WHILE` loop, and `WHILE` loops can be nested. The first of these complications can be handled in a way that would have been familiar to students if they had completed the implementation of a *BreakStatement* in Parva itself (a class exercise) and the second is best handled by parameterizing the *LabelledStatement* parser.

```
EndWhileStatement
= "ENDWHILE"                      (. if (loopCount == 0)
                                          SemError("only allowed to end a WHILE loop"); .)
  EOLS .
```