RHODES UNIVERSITY

November Examinations - 2004

Computer Science 301 - Paper 2

Examiners:
Prof P.D. Terry
Prof 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]

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

(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]

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

(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]

(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]

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]

(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]

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

(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]

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]

(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]

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]

(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)
}
```

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.

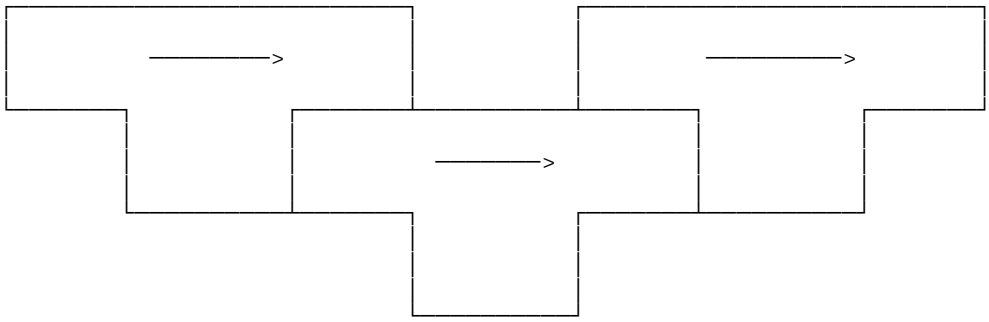
Save the honour of the Department! Spend the next 24 hours using Coco/R to develop a subset FORTRAN compiler that targets the PVM and handles the set of statements loosely described above, and then present a report and a Cocol grammar showing how you would do this. To assist you in this task we shall 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.

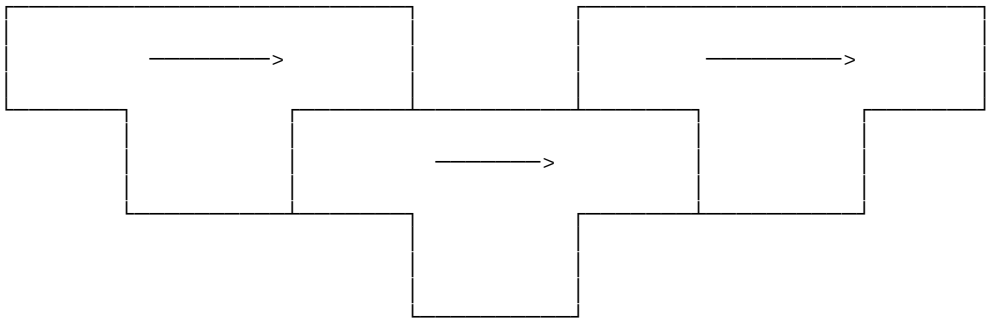
Computer Science 301 - November 2004

T Diagrams for question A2 STUDENT NUMBER

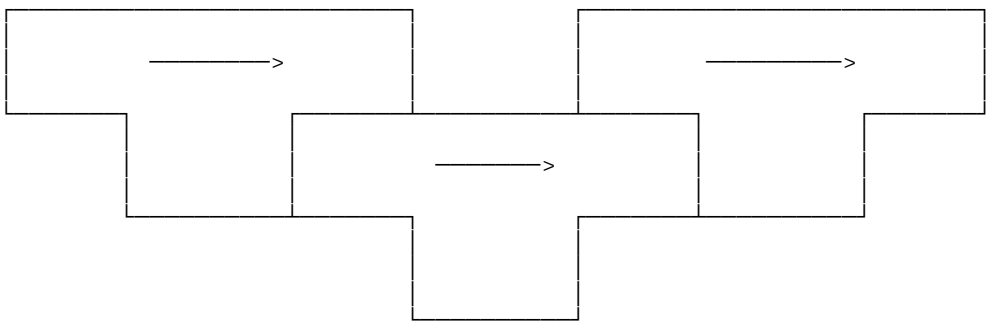
Hand this page in with your answer book. Make sure that you have added your student number.

Part A2(b)





Part A2(c)



Attributed grammar for question A3 STUDENT NUMBER

Hand this page in with your answer book. Make sure that you have added your student number.

```
COMPILER EBNF $CN
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.
```