RHODES UNIVERSITY

June Examinations - 2002

Computer Science 301 - Paper 1

Examiners:
Prof P.D. Terry
Prof E.H. Blake

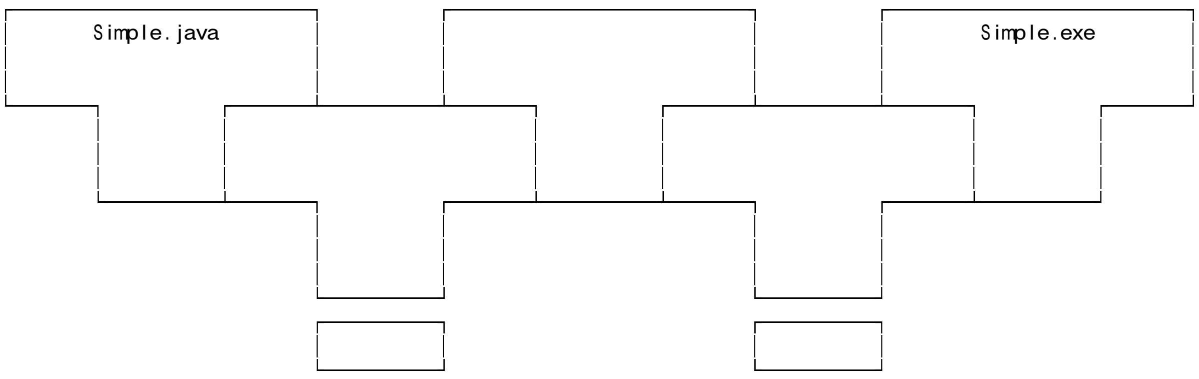
Time 3 hours
Marks 180
Pages 9 (please check!)

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 [90 marks]

1. A Java compiler might typically achieve its goals by compiling first to so-called Java byte code (an interpretable pseudo-assembly code) and then using a complementary JIT ("just in time") assembler to finish the job. Complete the T diagram representation of how such an arrangement would handle the compilation of a simple Java program. [8 marks]



2. Formally, a grammar G is a quadruple $\{ N, T, S, P \}$ with the four components

   (a) N - a finite set of **non-terminal** symbols,
   (b) T - a finite set of **terminal** symbols,
   (c) S - a special **goal** or **start** or **distinguished** symbol,
   (d) P - a finite set of **production rules** or, simply, **productions**.

   where a production relates to a pair of strings, say α and β, specifying how one may be transformed into the other:

   $$\alpha \rightarrow \beta \text{ where } \alpha \in (N \cup T)^* N (N \cup T)^* \text{ , } \beta \in (N \cup T)^*$$

   and formally we can define a language $L(G)$ produced by a grammar G by the relation

   $$L(G) = \{ \sigma \mid \sigma \in T^* ; S \Rightarrow^* \sigma \}$$

   In terms of this notation, express **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by [3 marks each]

   (a) A context-sensitive grammar

   (b) A context-free grammar

   (c) Reduced grammar

(d) FIRST(A) where $A \in N$

(e) FOLLOW(A) where $A \in N$

(f) Nullable productions

3. Long ago, the Romans represented 1 through 10 as the strings

```
I   II   III   IV   V   VI   VII   VIII   IX   X
```

The following grammar attempts to recognize a sequence of such numbers, separated by commas and terminated by a period:

```
COMPILER Roman
  PRODUCTIONS
    Roman      = Number { "," Number }  "."  EOF .
    Number     = StartI | StartV | StartX .
    StartI     = "I" ( "V" | "X" | [ "I" [ "I" ] ] ) .
    StartV     = "V" [ "I" ] [ "I" ] [ "I" ] .
    StartX     = "X" .
END Roman.
```

(a) What do you understand by the concept of an ambiguous grammar? [2 marks]

(b) Why is this particular grammar ambiguous? [2 marks]

(c) What do you understand by the concept of equivalent grammars? [2 marks]

(d) Give an equivalent grammar to the one above but which is unambiguous. [4 marks]

(e) Even though the grammar above is ambiguous, develop a matching hand-crafted recursive descent parser for it, similar to those that were developed in the practical course. The skeleton of such a system, including the code for a suitable scanner, is provided in an appendix to this paper. You need not modify the code that is already there; simply write out the parser functions. Your parser should detect and report errors, but there is no need to incorporate "error recovery". [10 marks]

(f) If the grammar is ambiguous (and thus cannot be LL(1)) does it follow that your parser would fail to recognize a correct sequence of Roman numbers, or would report success for an invalid sequence? Justify your answer. [2 marks]

4. In the compiler studied in the course the following Cocol code was used to handle code generation for a simple *IfStatement*:

```
IfStatement
=                     (. CGEN_labels testlabel; .)
   "if"
   "(" Condition ")"  (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Block              (. CGen->backpatch(testlabel); .) .
```

Suppose it was required to extend the system to provide an optional *else* clause:

```
Ifstatement = "(" Condition ")" Block [ "else" Block ] .
```

How would the Cocol specification have to be changed to generate efficient code for this extension? [6 marks]

5. In the compiler studied in the course the following Cocol code was used to handle code generation for a *WhileStatement*:

```
WhileStatement
=                     (. CGEN_labels whilelabel, testlabel, endlabel; .)
   "while"            (. CGen->storelabel(whilelabel); .)
   "(" Condition ")"  (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Block              (. CGen->jump(endlabel, whilelabel);
                         CGen->backpatch(testlabel); .) .
```

This generates code that matches the outline

```
whilelabel:   Condition
testlabel:    BZE  endlabel
              Block
              BRN  whilelabel
endlabel:
```

Some authors contend that it would be preferable to generate code that matches the outline

```
whilelabel:   BRN  testlabel
looplabel:    Block
testlabel:    Condition
              BNZ  looplabel
endlabel:
```

Their argument is that in most situations a loop body is executed many times, and that the efficiency of the system will be markedly improved by executing only one branch instruction on each iteration.

(a) Do you think the claim is justified for an interpreted system such as we have used? Explain your reasoning. [3 marks]

(b) If the suggestion is easily implementable in terms of our code generating functions, show how this could be done. If it is not easily implementable, why not? [3 marks]

6. The following Cocol description is of a set of letters in envelopes ready to take to the post office.

```
COMPILER Mail $XCN
/* Describe simple set of mailing envelopes */

IGNORE CHR(1) .. CHR(13)

CHARACTERS
  eol       = CHR(13) .
  digit     = "0123456789" .
  inaddress = ANY - CHR(0) - '$:' - eol .
  sp        = CHR(32) .

TOKENS
  number    = "postcode:" sp digit { digit } .
  info      = inaddress { inaddress } .

PRODUCTIONS
  Mail      = { Envelope } EOF .
  Envelope  = Stamp { Stamp } Person Address .
  Stamp     = "$1" | "$2" | "$3" .         /* values of stamps implied */
  Address   = Street Town [ PostCode ] .
  Person    = info .
  Street    = info .
  Town      = info .
  PostCode  = number .
END Mail.
```

What would you have to add to this grammar so that the parser system could tell you (a) the total value of the stamps on all the envelopes (make the obvious assumption about the value of a stamp denoted by $x being the number x) (b) the names of all people whose addresses did **not** contain postcodes? For your (and Postman Pat's) convenience the grammar has been spread out on the last page of this question paper, which you may detach and submit with your answer book. [10 marks]

7. The following Parva program exemplifies two oversights of the sort that frequently trouble beginner programmers - the array `list` has been declared, but never referenced, while the variable `total` has been correctly declared, but has not been defined (initialised) before being referenced.

```
void main () {
  int item, total, list[10];
  read(item);
  while (item != 0) {
    if (item > 0) { total = total + item; }
    read(item);
  }
  println("total of positive numbers is ", total);
}
```

Discuss the extent to which these "errors" might be detected by suitable extensions of the Parva compiler/interpreter system developed in this course. You do not need to give the algorithms in detail, but you might like to structure your answer on the following lines: [6 marks]

Variables declared but never referenced:

	Not detectable at compile time because ....
or	Detectable at compile time by ....
and/or	Not detectable at run time because ....
or	Detectable at run time time by ....

Variables referenced before their values are defined:

	Not detectable at compile time because ....
or	Detectable at compile time by ....
and/or	Not detectable at run time because ....
or	Detectable at run time time by ....

8. (a) Briefly clarify what you understand by the terms "scope/visibility" and "existence" as they apply to the implementation of variables for block-structured imperative languages such as Parva, Java, C++, Pascal or Modula-2. [5 marks]

(b) Briefly describe a suitable mechanism that could be used for symbol table construction to handle scope rules and offset addressing for variables in a language like Parva. Illustrate your answer by giving a snapshot of the symbol table at each of the points indicated in the code below. (The first one has been done for you.) [6 marks]

```
void main () {
  int list[4], i, j, k;    // compilation point 1

  if (i > 0) {
    int a, b, c;           // compilation point 2

  } else {
    int c, a, d;           // compilation point 3

  }
  int b[3], last;          // compilation point 4
}
```

Point 1

Name	list	i	j	k							
Offset	1	6	7	8							

(c) If the declaration at point 3 were changed to

```
    int c, a, i;           // compilation point 3
```

then the code would be acceptable to a C++ compiler but not to a Parva or Java compiler, as C++ allows programmers much greater freedom in reusing/redeclaring identifiers in inner scopes. What benefits do you suppose language designers would claim for either greater or reduced freedom? [3 marks]

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 to the blank diskette that will be provided.

During the translators course we have made frequent use of the Cocol/EBNF notation for expressing production rules, and in particular the use of the {} and [] meta-brackets introduced by Wirth in his 1977 paper. An example of a system of productions using this notation is as follows:

```
Home      = Family { Pets } [ Vehicle ] "house" .
Pets      = "dog" [ "cat" ] | "cat" .
Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
Family    = Parents { Children } .
Parents   = "Dad" "Mom" | "Mom" "Dad" .
Parents   = "Dad" | "Mom" .
Child     = "Margaret" | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
```

In analysing such productions and/or testing the grammar it has often been suggested that:

(a) they be rewritten without using the Wirth brackets, but using right recursive productions and an explicit ε, as for example

```
Home      = Family AllPets Transport "house" .
AllPets   = Pets AllPets | ε .
Transport = Vehicle | ε .
Pets      = "dog" PussyCat | "cat" .
PussyCat  = "cat" | ε .
Vehicle   = ( "scooter" "bicycle" ) | "fourbyfour" .
Family    = Parents Offspring .
Offspring = Offspring Children | ε .
Parents   = "Dad" "Mom" | "Mom" "Dad" .
Parents   = "Dad" | "Mom" .
Child     = "Margaret" | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
```

(b) a check be made to see that all the non-terminals have been defined (this does not occur in the above case - `Children` is undefined);

(c) a check be made to see that each non-terminal is defined only once (this does not occur in the above case - there are two rules for `Parents`);

(d) a check be made to see that each non-terminal (other than the goal) must appear on the right side of at least one production (this does not occur in the above case; `Child` defines a non-teminal which does not appear in any other production).

As a useful service to others who might take this course in future years (and hopeful that, although you might encourage others to do so, you are not forced to do so yourself!), develop a system using Coco/R that will carry out the above transformations and checks.

Here are some suggestions for deriving a complete system:

(a) In the exam kit (`EXAM.ZIP`) you will find the executables and support files for Coco/R, as used in the practical course. There is also the skeleton of a grammar file `EBNF.ATG` with suitable definitions of character sets and tokens similar to those you have seen before.

(b) It should be obvious that you will need to set up a "symbol table". In the exam kit is supplied the skeleton of an "include file" `EBNF.H`, which has a rudimentary form of table that you might like to extend. The template list handler file `LIST.H` familiar from earlier practicals has also been supplied.

(c) Inventing additional non-terminal names (without clashing with others that might already exist) might be done with the aim of deriving, for example

```
Home = Family HomeSeq1 HomeOpt2 "house" .
```

(d) In the exam kit will be found some other example data and production sets to assist in your development of the system, and an executable derived from a model solution.

(e) After converting a set of productions from EBNF to BNF, try converting the BNF productions once again to check that your system works consistently.

```
COMPILER EBNF $XCN
/* Convert a set of EBNF productions to use BNF conventions, and carry out
   some rudimentary checks on their being properly defined.
   YOUR IDENTITY HERE */

#include "ebnf.h"

CHARACTERS
  cr       = CHR(10) .
  lf       = CHR(13) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_".
  digit    = "0123456789" .
  noquote1 = ANY - "'" - cr - lf .
  noquote2 = ANY - '"' - cr - lf .

IGNORE CHR(9) .. CHR(13)

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

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

PRODUCTIONS
  EBNF
  =                                    (. TABLE_InitTable(); .)
    { Production }
    EOF
    .

  Production
  =                                    (. char Name[100];
                                          int i; .)
    SYNC nonterminal                   (. LexString(Name, sizeof(Name) - 1);
                                          TABLE_Add(Name, i);
                                       .)
    WEAK "="                           /* obviously more needed here */
    SYNC "."
    .

END EBNF.
```

```
// Various common items for grammar handler

#ifndef EBNF_H
#define EBNF_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define  boolean  int
#define  bool     int
#define  true     1
#define  false    0
#define  TRUE     1
#define  FALSE    0
#define  maxint   INT_MAX

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  define  pathsep '\\'
#else
#  define  pathsep '/'
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{ int i;
  char old[256];
  strcpy(old, oldstr);
  i = strlen(old);
  while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
  if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
  if (ext[0] == '.') sprintf(newstr,"%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

typedef struct {
  char name[50];
  // you may need other members
} ENTRIES;

const int TABLE_MaxEntries = 1000;            // limit on table size
static int TABLE_NEntries;                    // number of active entries
static ENTRIES TABLE_Table[TABLE_MaxEntries]; // the table itself

static void TABLE_InitTable() {
// Initialise table
  TABLE_NEntries = 0;
}

static void TABLE_Add (char *name, int &position) {
// Attempt to add name to the table, and return position as the
// index where it was either added previously or added by this call
  if (TABLE_NEntries > TABLE_MaxEntries) {
    fprintf(stderr, "Table Overflow"); exit(1);
  }
  strcpy(TABLE_Table[TABLE_NEntries + 1].name, name); // sentinel
  position = 1;
  while (strcmp(name, TABLE_Table[position].name)) position++;
  if (position > TABLE_NEntries) TABLE_NEntries++;    // new entry
}

static void TABLE_ListProductions() {
// List all entries to standard output file
  for (int i = 1; i <= TABLE_NEntries; i++)
    printf("%s\n", TABLE_Table[i].name);
  printf("\n");
}

#endif /* EBNF_H */
```

Question 3 - Scanner and Driver code for Roman Number recogniser

(There is no need to modify any of this code)

```
// Roman Numbers Parser
// Usage: Roman DataFile

#include "misc.h"
#include "set.h"

// +++++++++++++++++++++++++ Character Handler ++++++++++++++++++++++++

FILE *inFile;
char ch;                        // Lookahead character for scanner

void GetChar (void) {
// Obtains next character ch from inFile, or CHR(0) if EOF reached
  ch = fgetc(inFile); putchar(ch);
  if (ch == EOF) ch = '\0';
}

// +++++++++++++++++++++++++ Scanner ++++++++++++++++++++++++++++++++++

enum SYMBOLS { EOFSym, unknownSym, ISym, VSym, XSym, commaSym, periodSym };
SYMBOLS sym;                    // Lookahead token for parser

void GetSym (void) {
// Scans for next sym from inFile
  while (ch > 0 && ch <= ' ') GetChar(); // skip whitespace
  switch (toupper(ch)) {
    case '\0' : sym = EOFSym; break;
    case 'I'  : sym = ISym; GetChar(); break;
    case 'V'  : sym = VSym; GetChar(); break;
    case 'X'  : sym = XSym; GetChar(); break;
    case ','  : sym = commaSym; GetChar(); break;
    case '.'  : sym = periodSym; GetChar(); break;
    default   : sym = unknownSym; GetChar(); break;
  }
}

// +++++++++++++++++++++++++ Parser +++++++++++++++++++++++++++++++++++

void Abort (char *S) {
// Abandon parsing with error message S
  fprintf(stderr, "%s", S); exit(1);
}

void accept (SYMBOLS WantedSym, char *S) {
// Check that lookahead token is WantedSym
  if (sym == WantedSym) GetSym(); else Abort(S);
}

void main (int argc, char *argv[])  {
  if ((inFile = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "Could not open %s", argv[1]); exit(1);
  }
  GetChar();               // Initialise character handler
  GetSym();                // Initialise scanner
  Roman();                 // Initialise parser
  fprintf(stderr, "Parsed correctly");
}
```

Question 6 - Postman Pat (Detach this page and complete your solution on it)

```
COMPILER Mail $XCN
/* Describe simple set of mailing envelopes */
/* INSERT YOUR IDENTITY HERE ------------------------ > */

IGNORE CHR(1) .. CHR(13)

CHARACTERS
  eol       = CHR(13) .
  digit     = "0123456789" .
  inaddress = ANY - CHR(0) - '$:' - eol .
  sp        = CHR(32).

TOKENS
  number    = "postcode:" sp digit { digit } .
  info      = inaddress { inaddress } .

PRODUCTIONS
  Mail
  =
    {
    Envelope
    } EOF
  .

  Envelope
  =
    Stamp
    { Stamp
    } Person
    Address
  .

  Stamp
  =
      "$1"
    | "$2"
    | "$3"
  .

  Address
  =
    Street
    Town
    [ PostCode
    ]
  .

  Person
  =
    info
  .

  Street
  =
    info
  .

  Town
  =
    info
  .

  PostCode
  =
    number
  .

END Mail.
```