RHODES UNIVERSITY

November Examinations - 2003

Computer Science 301 - Paper 2

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

Time 3 hours
Marks 180
Pages 10 (please check!)

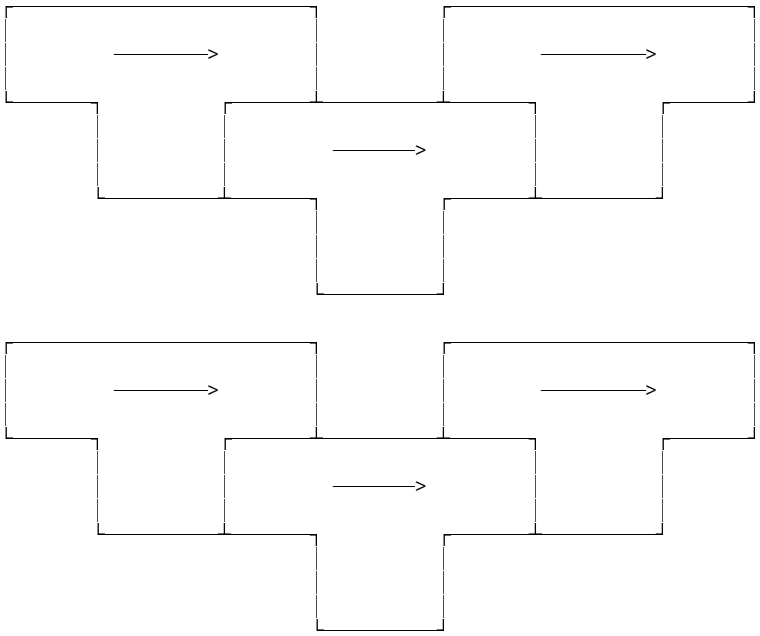
Answer all questions. Answers may be written in any medium except red ink.

(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, summaries of useful Java library classes, and the source listing of a Parva compiler. 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 [100 marks]

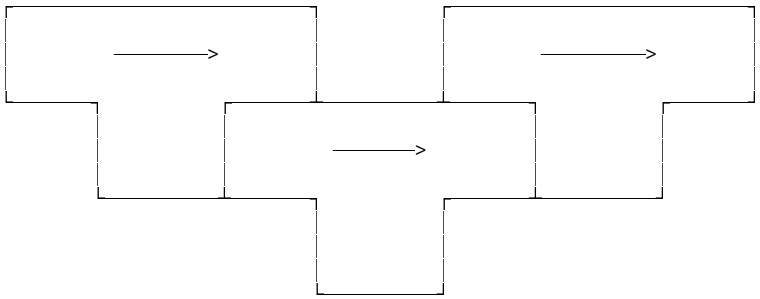
A1. (a) A pretty-printer is a form of compiler that will take a source program and reformat it to make it look ″pretty″, without actually changing the source in any way other than by inserting some spaces and line breaks and deleting other spaces and line breaks.

Suppose you are asked to write a pretty-printer for Parva, using the familiar Java version of Coco/R to provide the scanner, parser and driver components of the system. Complete the T diagrams below to show the steps you would follow to do this. *(A copy of these T-diagrams appears as an attachment to this paper, which you can detach and hand in with your answer book.)* [6 marks]



(b) Do you suppose a pretty-printer would need to make use of a symbol table handler? Give a reason for your answer. [3 marks]

(c) Develop a further T diagram to show how the pretty-printer would be applied to process a Parva program like `SAMPLE01.PAV`. [3 marks]



A2. Formally, a grammar G is defined by 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:

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

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

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

(a) In terms of this style of notation, define **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by [2 marks each]

(1) FIRST(σ) where $\sigma \in (N \cup T)^+$

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

(b) State the two rules that must be satisfied by the productions of the grammar in order for it to be classified as an LL(1) grammar. [6 marks]

(c) The Cocol grammar below (`RE.ATG`) describes a sequence of Regular Expressions (written one to a line) using the conventions discussed during the course.

```
COMPILER RE   /* Regular expression grammar */

CHARACTERS
  lf       = CHR(10) .
  control  = CHR(1) .. CHR(31) .
  noquote1 = ANY - control - "'" .
  noquote2 = ANY - control - '"' .
  meta     = "()*|[]-?+" .
  simple   = ANY - control - "'" - '"' - meta .

IGNORE  CHR(1) .. CHR(9) + CHR(11) .. CHR(31)

TOKENS
  atomic  = simple .
  escaped = "'" noquote1 "'" | '"' noquote2 '"' .
  EOL     = lf .

PRODUCTIONS
  RE         = { Expression EOL } EOF .
  Expression = Term { "|" Term } .
  Term       = Factor { Factor } .
  Factor     = Element [ "*" | "?" | "+" ] .
  Element    = Atom | Range | "(" Expression ")" .
  Range      = "[" OneRange { OneRange } "]" .
  OneRange   = Atom [ "-" Atom ] .
  Atom       = atomic | escaped .
END RE.
```

Determine the following FIRST and FOLLOW sets: [8 marks]

FIRST(*Factor*)	FOLLOW(*Factor*)
FIRST(*OneRange*)	FOLLOW(*OneRange*)

(d) Does the production for *Factor* lead to LL(1) errors? Justify your answer. [3 marks]

A3. Chomsky classified grammars into four types, which he called 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 numbers corresponding to the other types? [3 marks]

Your answer should take the form

A Regular grammar is also called …
A Context-free grammar is also called …
A Context-sensitive grammar is also called …

(b) Consider the following simple grammars defined on a vocabulary with two non-terminals { A, B }, four terminals { a, b, c, d }. Assume that A is the goal symbol.

The grammars are not supposed to describe any sensible languages, of course. Classify each grammar as being of one of the above types. [6 marks]

```
Grammar 1
    A   = "a" B | "b" .
    B   = "c" A | "d" .

Grammar 2
    A   = "a" B | A B "b" .
    A B = "a" B "b" .
    B   = B "a" "b" "c" "d" .

Grammar 3
    A   = "a" A | "b" .
    B   = "b" A | "c" A "d" .
```

(c) Do any of the grammars incorporate "useless productions"? If so, which grammar(s) have useless productions, and which productions are the useless ones? [3 marks]

A4. You will recall that my brother occasionally appears on television in various roles. The following set of productions attempts to describe some exciting viewing on the SABC.

```
SABCTV      = { Programme } "Closedown" .
Programme   = "Announcement" { QualityShow  }
              [ "Reminder" ] /* you know it's the right thing to do */ .
QualityShow = ( "Frasier" | "MyFamily" | "Generations" ) { Advert } .
Advert      = "CTMTiles" | "Domestos" | "VodaCom" | "Nando's" | "LGDigital" .
```

Assuming that you have available a suitable scanner method called `getSym` that can recognize the terminals of this language and classify them appropriately as members of an enumeration

```
EOFSym, noSym, closeSym, announceSym, remindSym, frasierSym,
myFamilySym, generationsSym, ....
```

develop a hand-crafted recursive descent **parser** for watching a typical boring evening on television. Your parser can take drastic action if an invalid sequence is detected - simply produce an appropriate error message and then terminate parsing (in effect, switch off the television set). (*You are not required to write any code to implement the getSym method, nor need you take any precautions to check on my brother's appearances*.) [20 marks]

A5. Consider the grammar given in question A2 for describing regular expressions. For this next question ignore any potential LL(1) problems with the grammar.

How would you add appropriate actions and attributes so that you could generate a program that would parse a sequence of regular expressions and report on the **alphabets** used in each one. For example, given input like

```
a | b c d | ( x y z )*
[a-g A-G] [x - z]?
a? "'" z+
```

the output should be something like

```
Alphabet = a b c d x y z
Alphabet = A B C D E F G a b c d e f g x y z
Alphabet = ' a z
```

A machine readable version of the grammar can be found in the exam kit (`RE.ATG`), should you prefer to develop a machine readable solution. A printed version of the productions is given as an attachment. [24 marks]

Hints:

It will be simplest to use a simple static array to store each alphabet.

"Keep your grammar as simple as possible, but no simpler".

Pay particular attention to where you introduce the actions/attributes; do not simply write them in random positions in the grammar.

A6. Consider the following small Parva program

```
int a, b = 12;

void two(int x) {
  // here ------------------------------
  int d = x + 2;
  if (x > 1) two(x-1);
  write(d);
}

void one(c) {
  int x, y, z;
  two(4);
}

void main() {
  one(b);
}
```

(a) The programmer has introduced the identifier `x` in more than one function. Does this represent an error? Explain your answer. [3 marks]

(b) Assuming that the program is, in fact, correct, show that you understand the general concept of using activation records to implement function calls, by drawing a diagram showing what these records would look like at run-time when the point marked `// here` is reached for the second time. [8 marks]

Section B [80 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.

A minor crisis has developed in the Computer Science department. This year, as an experiment, the first year students have been taught programming using the influential Parva language. This has proved eminently successful, and the students are due to write a practical examination on Monday. Today is Sunday. Earlier this morning the lecturer in charge of the class discovered, to her horror, that the existing Parva compiler makes provision only for integer and boolean data types, and not for the character type that would be required if students were asked to write code like the following:

```
void main () {
// Read a sentence and write it backwards
  char[] sentence = new char[1000];
  int i = 0;
  char ch;
  read(ch);
  while (ch != '.') {  // input loop
    sentence[i] = ch;
    i = i + 1;
    read(ch);
  }
  while (i > 0) {      // output loop
    i = i - 1;
    write(sentence[i]);
  }
}
```

or similar applications which require simple character manipulations, including the ability to mix integers and characters appropriately in expressions, detect type incompatibilities, cast between integers and characters when required, increment and decrement character variables and so on.

She had intended to set a simple problem in which the students were required to print a set of multiplication tables. When testing the solution, she made a typing error and submitted the following code to the compiler, with rather unexpected results:

```
void main () {
// Print a set of multiplication tables
  int i, j;
  for i = 1 to 10 do { // <----------------- should be j not i
    for i = 1 to 10 do write (i * j, "\t");
    write("\n");
  }
}
```

But worse is still to come! The lecturer has also been under the illusion that the key word *const* is used in the sense that it is used in C# and the word *final* is used in Java, namely as a modifier in a variable declaration that indicates that when a variable is declared it can be given a value which can then not be modified later. On checking the grammar for Parva she comes across the productions

```
Statement        = Block | ConstDeclarations | VarDeclarations | ...
ConstDeclarations = "const" OneConst { "," OneConst } ";" .
OneConst         = identifier "=" Constant .
Constant         = number | charLit | "true" | "false" | "null" .
VarDeclarations  = Type OneVar { "," OneVar } ";" .
OneVar           = identifier [ "=" Expression ] .
```

which she realizes will not correctly handle code of the form she needs, like

```
const int max = 2;
int i = 5;
const int iPlusMax = i + max;
```

In desperation she turns to the members of the third year class, imploring them to come up with a reliable new

version of the Parva compiler in 24 hours that will provide the character type, treat the keyword *const* as she intends, and detect situations where attempts are made to alter a for loop control within the loop body. She offers to provide them with a few examples of the sort of code that she expects the compiler to be able to handle. For example, besides the example involving character types, she provides a commented test program:

```
void main () {
// Rather meaningless program to illustrate variable declarations
// that incorporate a const modifier
  const int i = 10;                 // acceptable
  const char terminator;            // unacceptable
  const int[] list = new int[i];    // acceptable
  int j = i;
  while (j < 100) {
    const int k = j;                // acceptable
    read(i);                        // unacceptable
    i = j;                          // unacceptable
    list[4] = 5;                    // acceptable
    j = j + list[4];                // acceptable
    list = new int[12];             // unacceptable
  }
}
```

Rise to the challenge! Produce a new compiler, and avoid a disaster for the department.

Note:

(a) In the exam kit (`EXAM.ZIP`) you will find the executables and support files for Coco/R, as used in the practical course. You will also find the complete attribute grammar and support files for the Parva language as developed by the end of your lecture and practical course.

(b) In the exam kit will be found some other example programs like those above to assist in your understanding of the requirements and your development of the system, and an executable derived from a model solution that you are free to use to compile these example programs or any others you may devise.

(c) Depending on your approach, your solution may require modifications to any or all of the grammar and support files in the exam kit.

(d) It is not particularly difficult to provide "first approximations" to these extensions and modifications. The examiners will, however, be looking for maturity in your solution and, in particular for signs that you have seen past the obvious "quick fix".

Summary of useful library classes

```
class SymSet  { // simple set handling routines
  public SymSet()
  public SymSet(int[] members)
  public boolean equals(Symset s)
  public void incl(int i)
  public void excl(int i)
  public boolean contains(int i)
  public boolean isEmpty()
  public int members()
  public SymSet union(SymSet s)
  public SymSet intersection(SymSet s)
  public SymSet difference(SymSet s)
  public SymSet symDiff(SymSet s)
  public void write()
  public String toString()
} // SymSet

public class OutFile {  // text file output
  public static OutFile StdOut
  public static OutFile StdErr
  public OutFile()
  public OutFile(String fileName)
  public boolean openError()
  public void write(String s)
  public void write(Object o)
  public void write(int o)
```

```
    public void write(long o)
    public void write(boolean o)
    public void write(float o)
    public void write(double o)
    public void write(char o)
    public void writeLine()
    public void writeLine(String s)
    public void writeLine(Object o)
    public void writeLine(int o)
    public void writeLine(long o)
    public void writeLine(boolean o)
    public void writeLine(float o)
    public void writeLine(double o)
    public void writeLine(char o)
    public void write(String o,  int width)
    public void write(Object o,  int width)
    public void write(int o,     int width)
    public void write(long o,    int width)
    public void write(boolean o, int width)
    public void write(float o,   int width)
    public void write(double o,  int width)
    public void write(char o,    int width)
    public void writeLine(String o,  int width)
    public void writeLine(Object o,  int width)
    public void writeLine(int o,     int width)
    public void writeLine(long o,    int width)
    public void writeLine(boolean o, int width)
    public void writeLine(float o,   int width)
    public void writeLine(double o,  int width)
    public void writeLine(char o,    int width)
    public void close()
  } // OutFile

  public class InFile {    // text file input
    public static InFile StdIn
    public InFile()
    public InFile(String fileName)
    public boolean openError()
    public int errorCount()
    public static boolean done()
    public void showErrors()
    public void hideErrors()
    public boolean eof()
    public boolean eol()
    public boolean error()
    public boolean noMoreData()
    public char readChar()
    public void readAgain()
    public void skipSpaces()
    public void readLn()
    public String readString()
    public String readString(int max)
    public String readLine()
    public String readWord()
    public int readInt()
    public long readLong()
    public int readShort()
    public float readFloat()
    public double readDouble()
    public boolean readBool()
    public void close()
  } // InFile

  class ArrayList {   //  Maintenance of simple lists of objects
    public ArrayList()
    public void clear()
    public int size()
    public boolean isEmpty()
    public void add(Object o)
    public Object get(int index)
    public Object remove(int index)
  } // ArrayList
```

Strings and Characters in Java

The following rather meaningless program illustrates various of the string and character manipulation methods that are available in Java and which will be found to be useful in developing translators.

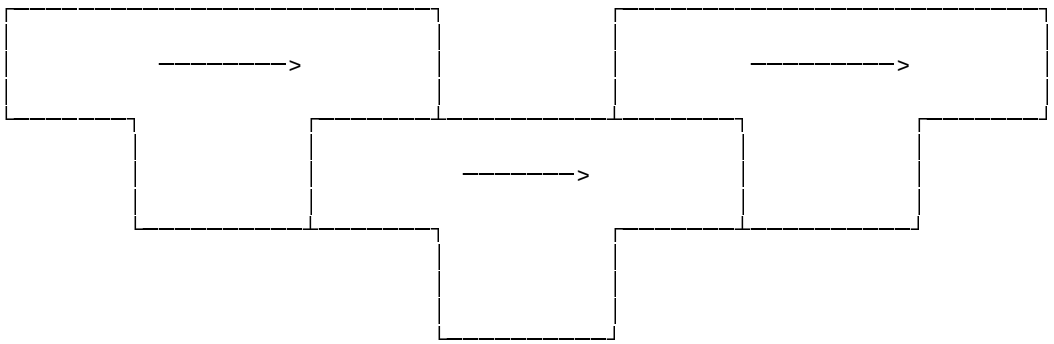
```
import java.util.*;

class demo {
  public static void main(String[] args) {
    char c, c1, c2;
    boolean b, b1, b2;
    String s, s1, s2;
    int i, i1, i2;

    b = Character.isLetter(c);             // true if letter
    b = Character.isDigit(c);              // true if digit
    b = Character.isLetterOrDigit(c);      // true if letter or digit
    b = Character.isWhitespace(c);         // true if white space
    b = Character.isLowerCase(c);          // true if lowercase
    b = Character.isUpperCase(c);          // true if uppercase
    c = Character.toLowerCase(c);          // equivalent lowercase
    c = Character.toUpperCase(c);          // equivalent uppercase
    s = Character.toString(c);             // convert to string
    i = s.length();                        // length of string
    b = s.equals(s1);                      // true if s == s1
    b = s.equalsIgnoreCase(s1);            // true if s == s1, case irrelevant
    i = s1.compareTo(s2);                  // i = -1, 0, 1 if s1 < = > s2
    s = s.trim();                          // remove leading/trailing whitespace
    s = s.toUpperCase();                   // equivalent uppercase string
    s = s.toLowerCase();                   // equivalent lowercase string
    char[] ca = s.toCharArray();           // create character array
    s = s1.concat(s2);                     // s1 + s2
    s = s.substring(i1);                   // substring starting at s[i1]
    s = s.substring(i1, i2);               // substring s[i1 ... i2]
    s = s.replace(c1, c2);                 // replace all c1 by c2
    c = s.charAt(i);                       // extract i-th character of s
//    s[i] = c;                            // not allowed
    i = s.indexOf(c);                      // position of c in s[0 ...
    i = s.indexOf(c, i1);                  // position of c in s[i1 ...
    i = s.indexOf(s1);                     // position of s1 in s[0 ...
    i = s.indexOf(s1, i1);                 // position of s1 in s[i1 ...
    i = s.lastIndexOf(c);                  // last position of c in s
    i = s.lastIndexOf(c, i1);              // last position of c in s, <= i1
    i = s.lastIndexOf(s1);                 // last position of s1 in s
    i = s.lastIndexOf(s1, i1);             // last position of s1 in s, <= i1
    i = Integer.parseInt(s);               // convert string to integer
    i = Integer.parseInt(s, i1);           // convert string to integer, base i1
    s = Integer.toString(i);               // convert integer to string

    StringBuffer                           // build strings
      sb = new StringBuffer(),             //
      sb1 = new StringBuffer("original");  //
    sb.append(c);                          // append c to end of sb
    sb.append(s);                          // append s to end of sb
    sb.insert(i, c);                       // insert c in position i
    sb.insert(i, s);                       // insert s in position i
    b = sb.equals(sb1);                    // true if sb == sb1
    i = sb.length();                       // length of sb
    i = sb.indexOf(s1);                    // position of s1 in sb
    sb.delete(i1, i2);                     // remove sb[i1 .. i2]
    sb.replace(i1, i2, s1);                // replace sb[i1 .. i2] by s1
    s = sb.toString();                     // convert sb to real string
    c = sb.charAt(i);                      // extract sb[i]
    sb.setCharAt(i, c);                    // sb[i] = c
  }
}
```

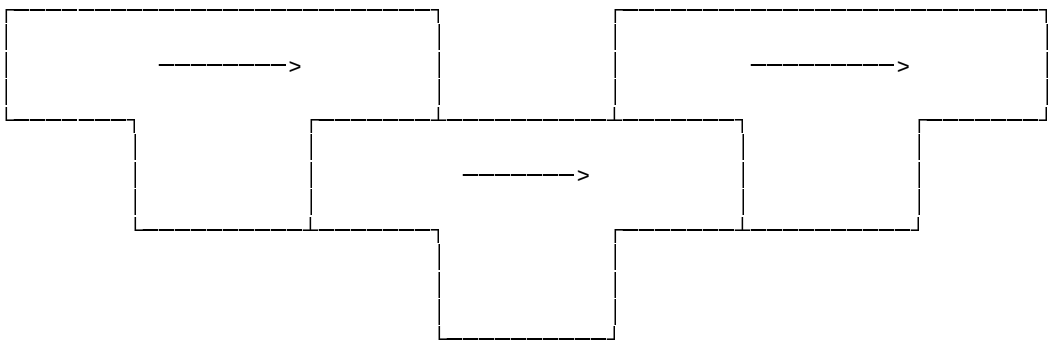
Computer Science 301 - November 2003

T Diagrams for question A1 STUDENT NUMBER

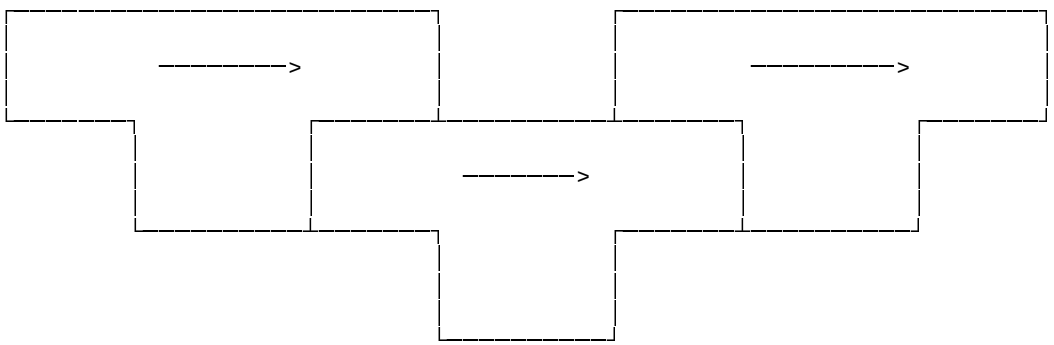
Hand this page in with your answer book. Make sure that you have added your student number.

Part A1(b)





Part A1(c)



Computer Science 301 - November 2003

Grammar for questions 2 and 5 STUDENT NUMBER

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

```
COMPILER RE



PRODUCTIONS
  RE
  = { Expression
      EOL
    } EOF .

  Expression
  = Term
    { "|" Term
    } .

  Term
  = Factor
    { Factor
    } .

  Factor
  = Element
    [   "*"
      | "?"
      | "+"
    ] .

  Element
  =   Atom
    | Range
    | "(" Expression
      ")" .

  Range
  = "["  OneRange
       { OneRange
       }
    "]" .

  OneRange
  = Atom
    [ "-" Atom
    ] .

  Atom
  =    atomic
    |  escaped
  .

END RE.
```