RHODES UNIVERSITY

November Examinations - 2006

Computer Science 301 - Paper 2

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

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

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

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

Section A [90 marks]

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

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

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

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

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

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

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

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

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

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

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

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

IGNORE control

PRODUCTIONS

  EBNF        = { Production } EOF .

  Production  = nonterminal "=" Expression  "." .

  Expression  = Term { "|" Term } .

  Term        = Factor { Factor } .

  Factor      =   nonterminal | terminal | "(" Expression ")"
                | "[" Expression "]" | "{" Expression "}" .

END EBNF.
```

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

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

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

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

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

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

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

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

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

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

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

Here are two attempts to write grammars describing an RPN expression:

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

and

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

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

(b) Using the expression

```
15  6  -  -
```

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

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

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

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

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

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

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

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

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

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

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

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

Section B [90 marks]

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

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

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

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

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

Parva: `(demo.pav)`

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

Java equivalent: `(demo.java)`

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

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

} // demo
```

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

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

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

(c) *class* declarations [10 marks]

(d) *const* declarations [16 marks]

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

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

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

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

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

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

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

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

Section C

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

A pretty-printer is a form of translator that takes source code and translates this into object code that is expressed in the same language as the source. This probably does not sound very useful! However, the main aim is to format the "object code" neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

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

```
Goal={One}.
One

= Two "plus" Four ".".

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

and produce the much neater output

```
Goal
  = { One } .

One
  = Two "plus" Four "." .

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

```
Five
  = Six  [ Six ] .

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

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

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

Section D

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

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

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

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

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

Free information

Context-free unattributed grammar for Parva

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

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

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

IGNORE CHR(9) .. CHR(13)

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

Summary of useful library classes

The following summarizes the simple set handling and I/O classes that have been useful in the development of applications using the Coco/R compiler generator.

```
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 are 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

      StringTokenizer                         // tokenize strings
        st = new StringTokenizer(s, ".,");    // delimiters are . and ,
        st = new StringTokenizer(s, ".,", true); // delimiters are also tokens
        while (st.hasMoreTokens())            // process successive tokens
          process(st.nextToken());
    }
  }
```

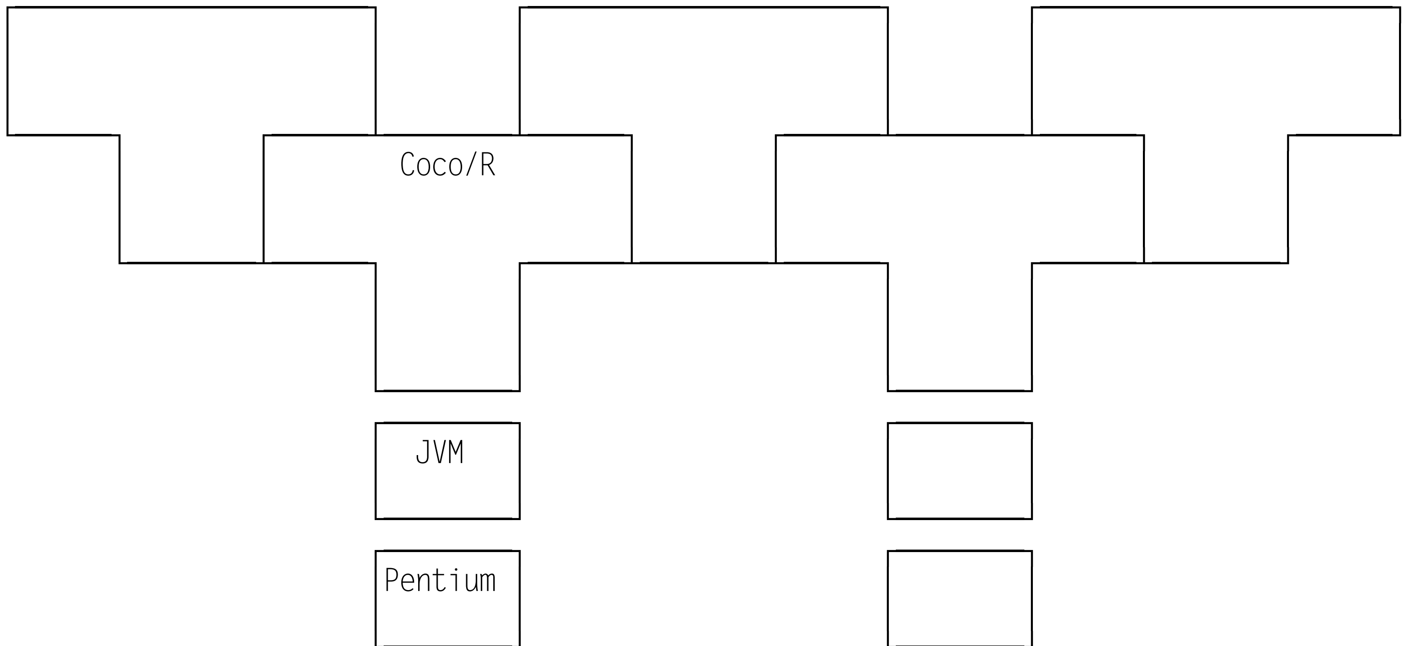
Simple list handling in Java

The following is the specification of useful members of a Java (1.4) list handling class useful in developing translators as discussed in this course. This class will "work" with Java 5.0, but the compiler will issue warnings, as `ArrayList` has been redefined to be a "generic" class.

```
  class ArrayList
  // Class for constructing a list of objects

    public ArrayList()
    // Empty list constructor

    public void add(Object o)
    // Appends o to end of list

    public void add(int index, Object o)
    // Inserts o at position index

    public Object get(int index)
    // Retrieves an object from position index

    public void clear()
    // Clears all elements from list

    public int size()
    // Returns number of elements in list

    public boolean isEmpty()
    // Returns true if list is empty

    public boolean contains(Object o)
    // Returns true if o is in the list

    public boolean indexOf(Object o)
    // Returns position of o in the list

    public Object remove(int index)
    // Removes the object at position index

  } // ArrayList
```

T Diagrams for question 12

Building the translator



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