RHODES UNIVERSITY

November Examinations - 2005

Computer Science 301 - Paper 2

Examiners:
Prof P.D. Terry
Dr 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 the full text of Section B, summaries of useful Java library classes, and a Parva grammar. 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]

A1. A compiler is often described as incorporating several "phases", as shown in the diagram below

```
Character Handling
        |
        v
 Lexical Analysis
        |
        v
 Syntax Analysis
        |
        v
Semantic Analysis
        |
        v
 Code Generation
```

These phases are often developed in conjunction with an Error Handler and a Symbol Table Handler.

(a) Annotate the diagram to suggest the changes that take place to the representation of a program as it is transformed from source code to object code by the compilation process. [4 marks]

(b) Suggest - perhaps by example - the sort of errors that might be detected in each of the phases of the compilation process. [4 marks]

A2. (a) The regular expression 1 (1 | 0)* 0 describes the binary representations of the non-zero even integers (2, 4, 6, 8 ...). Give a regular expression that describes the binary representations of non-negative integers that are exactly divisible by 4 (four), that is (0, 4, 8, 12, 16 ...) [3 marks].

(b) Currency values are sometimes expressed in a format exemplified by

R12,345,101.99

where, for large amounts, the digits are grouped in threes to the left of the point. Exactly two digits appear to the right of the point. The leftmost group might only have one or two digits. Either complete the Cocol description of a token that would be recognized as a currency value or, equivalently, use conventional regular expression notation to describe a currency value. [5 marks]

Your solution should not accept a representation that has leading zeroes, like R001,123.00.

```
CHARACTERS
   digit =

TOKENS
   currency =
```

A3. Consider the following simple grammar. It has one nonterminal A and two terminals (and).

A = "(" A ")" A | ε . (G1)

(a) Is G1 an LL(1) grammar? If not, why not? [4 marks]

(b) What language does G1 describe? (Write down a few strings derived from A and the pattern should be obvious.) [3 marks]

(c) Is G1 an ambiguous grammar? Explain your reasoning. [2 marks]

Now consider the following grammar that looks rather similar but does not have an ε:

A = "(" A ")" | "(" ")" | A A . (G2)

(d) Is G2 an LL(1) grammar? If not, why not? [2 marks]

(e) What language does G2 describe? (Write down a few strings derived from A and the pattern should be obvious.) [3 marks]

(f) Is G2 an ambiguous grammar? Explain your reasoning. [2 marks]

(g) Steven Spielberg is about to film his latest blockbuster "Elevator 9/11", in which a group of gorgeous Americans (guess what) are trapped in an elevator (a lift) hundreds of feet above the ground by a group of ugly ... (you guessed right again). He wants to make sure he understands how to arrange for the elevator to get stuck. (So would you if you had the chance to spend a few hours with some of the hunks and bimbos he has lined up to act for him).

Mr Spielberg has been reliably informed that some high-rise skyscrapers have elevators that are based partway up the building, say on level (floor) X. People entering one of these elevators have a choice of pushing an UP key or a DOWN key as many times as they like. If the elevator is behaving normally it can go up or down one level for each push of the appropriate key, but it cannot drop below level X, and at the end of the day it is supposed to have returned to level X again. Write down the productions for a simple grammar that will allow Mr S to recognize a sequence of commands that does not trap the elevator, such as

UP UP DOWN DOWN or UP UP DOWN UP DOWN UP DOWN DOWN

Unusually it is sequences that should be rejected by the grammar, such as

DOWN UP or UP UP DOWN

that are the ones that really interest Mr S. (Hint: Treat UP and DOWN as key words!) [4 marks]

A4. The following set of productions attempts to describe a mixed goods and passenger train:

```
Train = LocoSection [ GoodsSection ] HumanSection .
LocoSection = "loco" { "loco" } .
GoodsSection = "open" { "fuel" { "fuel" } "open" | "open" } .
HumanSection = "guard" | { "coach" } "brake" .
```

Assume 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 the following enumeration

```
EOFSym, noSym, locoSym, fuelSym, openSym, coachSym, guardSym, brakeSym
```

Develop a hand-crafted recursive descent parser for recognizing valid trains. Your parser can take drastic action if an invalid train 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.)* [20 marks]

A5. The university data processing division is faced with a problem. At short notice they have been asked

(a) to prepare a list of the surnames of the staff who have PhD degrees, and
(b) to determine the percentage of staff that have this qualification.

A data file is available with a list of staff which includes entries like

```
Mr Paddy O'Toole.
Ms Ima Raver, BA.
Prof Dedley Serious, BSc(Hons), MSc, PhD, PhD.
Dr Heilee-Unlykelee, PhD.
John Paul George Ringo Muggins.  Prof Jo Bloggs, BSc.
```

Develop an attributed Cocol grammar for solving this problem. Output for the data file above should be on the lines of

```
The following 2 staff (33%) have PhD degrees:

Serious
Heilee-Unlykelee
```

Assume that

(a) names follow the patterns suggested above (start with an uppercase letter, may contain other letters or hyphens or apostrophes),
(b) titles would be limited to Mr/Miss/Ms/Mrs/Dr/Prof,
(c) degrees would be limited to BA, BSc, BA(Hons), BSc(Hons), MA, MSc or PhD.

A skeleton of a Cocol description is given below. A machine readable version of this can be found in the exam kit (`STAFFLIST.ATG`), should you prefer to hand in a machine readable solution. [24 marks]

```
COMPILER StaffList

CHARACTERS

TOKENS

IGNORE

PRODUCTIONS

END StaffList.
```

Hints:

"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. (a) The Parva compiler studied in the course is an example of what is often called a "one pass incremental compiler" - one in which the various phases summarized in question A1 are interleaved, so that the compiler makes a single pass over the source text of a program and generates code as it does so, without building an explicit AST. Discuss briefly whether the same technique could be used to compile one class of a Java program (consider the features of Java and Parva that support the use of such a technique, and identify features that seem to act against it). [3 marks]

(b) The Parva compiler supports the integer and Boolean types. Integer constants (literals) are represented in the usual "decimal" system, for example 1234.

Suppose we wished to allow hexadecimal and binary representations of integer literals as well, for example 012FH and 01101%. Which of the phases of compilation identified in question A1 would this affect, which would it not affect, and why? [4 marks]

(c) The Cocol grammar for constructing the Parva compiler contains the productions below, which are used to handle the declaration (and possible initialization) of variables.

```
VarDeclarations<StackFrame frame>     (. int type; .)
=  Type<out type>
   OneVar<frame, type>
   { "," OneVar<frame, type> } ';" .

OneVar<StackFrame frame, int type>    (. int expType; .)
=                                     (. Entry var = new Entry(); .)
   Ident<out var.name>                (. var.kind = Entry.Var;
                                         var.type = type;
                                         var.offset = frame.size;
                                         frame.size++; .)
   [ AssignOp                         (. CodeGen.loadAddress(var); .)
     Expression<out expType>          (. if (!compatible(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.assign(var.type); .)
   ]                                  (. Table.insert(var); .)
.
```

The symbol table insertion for each variable is made at the end of the `OneVar` production. A student has queried whether this production could have been changed to read as follows, where the symbol table entry is made earlier. The lecturer, by way of reply, asked her to consider what the outcome would be of trying to compile the statement

```
int i = i + 1;
```

What would your considered reaction be? [3 marks]

```
OneVar<StackFrame frame, int type>    (. int expType; .)
=                                     (. Entry var = new Entry(); .)
   Ident<out var.name>                (. var.kind = Entry.Var;
                                         var.type = type;
                                         var.offset = frame.size;
                                         Table.insert(var);   // +++++ moved
                                         frame.size++; .)
   [ AssignOp                         (. CodeGen.loadAddress(var); .)
     Expression<out expType>          (. if (!compatible(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.assign(var.type); .)
   ]
.
```

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.

It had to happen! Over the last eight weeks a team of over 60 dynamic young programmers (DYPs) have been developing sophisticated applications in the new wonder language Parva. Like all DYPs, they have paid far too little attention to documenting what they have done. Unfortunately the day of reckoning is upon them: a directive has gone out from the Project Documentation Tyrant (PDT) that in a mere 24 hours time they will be required to provide a full set of documentation for their code, preferably in the form of a set of HTML (web) pages, one for each completed program.

Their PM (Project Manager) recently browsed through a text on Java and was struck by the existence of a utility called *JavaDoc*. This is a system program that can parse Java source code files and extract documentation from it, if such documentation is embedded in the files in a systematic way. Any "documentation" stored in a source code file must, of course, be distinguishable from the "real" source code which a compiler would process, so the system relies on extracting the documentation from comments - which, however, must not be confused with other commentary in the program. For the JavaDoc utility this is achieved by allowing commentary to take one of three forms:

```
/*  This is a typical comment in familiar form  */

//  This is a typical simple one-line comment in familiar form

/** This is a special JavaDoc comment, where the leading ** acts as
    a special marker to distinguish it from a simple comment
*/
```

Of course, JavaDoc comments would be ignored (along with other comments) if the programs were to be compiled by the standard Java compiler itself.

After consulting the PDT, the PM has asked that the team can meet the deadline by developing a utility called ParvaDoc that will do the same sort of thing for Parva. She realises that the ParvaDoc system can be generated by using Coco/R to create a "compiler" that will parse Parva programs - but rather than generate PVM code, this compiler will ignore most of the usual source code and concentrate almost entirely on certain of the comments. The PDT has agreed that the submission of an attributed Cocol grammar for the ParvaDoc utility will serve as proof that the deadline can be met.

Very generously, the PDT has supplied some examples of simple Parva programs marked up in this way, along with the form of output that he suggests should be generated - and, of course, he has also provided a simple Cocol version of the basic grammar for Parva, a copy of the Coco/R compiler generator, and a fully working executable for a Parva compiler. All this should help those 65 DYPs whose qualifications are at stake.

The crucial "ParvaDoc" comments are bound by structural rules of their own. For the task at hand, assume that they can appear only (a) immediately at the start of the code or (b) immediately at the start of a function or method. These comments are intended to document such things as the overall purpose of the application and/or its individual functions, the author, the version, the intent of the parameters or arguments, the value returned (for functions other than "void" ones), and possible references to other documents or resources.

Here is a typical example of ParvaDoc commentary for a function:

```
int ReduceLength(int[] list, int n, int length) {
/** @purpose      Reduces a list of integer numbers by removing every n-th number
    @param length specifies the original length of the list
    @return       length of reduced list
    @param n      specifies that every n-th number is to be removed
    @see          "Algorithms for a Special Sunday" (2005)
    @param list   is a reference to the list of numbers
    @version      1234
    @author       PDT
*/
  ...
}
```

As can be seen, the different components of the documentation are introduced by tags like `@purpose` and `@author`. Not all of these sections need be present, and they may be introduced in any order.

After processing the source file, the ParvaDoc system should produce output like

```
int ReduceLength (int[] list, int n, int length)

Purpose:
 Reduces a list of integer numbers by removing every n - th number

Author:
 PDT

Version:
 1234

References:
 "Algorithms for a Special Sunday" (2005)

Returns:
 length of reduced list

Parameters:
    length specifies the original length of the list

    n specifies that every n - th number is to be removed

    list is a reference to the list of numbers
```

in which we can note that the output consists of the function signature, followed by a sorted and nicely formatted list of the components forming the documentation. (A more extensive set of examples of input and output can be found in the files that have been provided by the PDT.)

The system should provide some minimal checking - for example, that tags like `@param` and `@return` are not applied to an introductory ParvaDoc comment describing an application as a whole, and that the identifiers quoted at the start of any `@param` clauses match those in the function signature. An example of the output from an attempt to analyse an incorrectly documented application might be

```
      1 /**
      2     @author  P.D. Terry, Rhodes University, 2005
      3     @return  Tomorrow's oil price
   ****     ↑ not applicable here
      4     @param   Humidity in Gulf of Mexico
   ****     ↑ not applicable here
      5 */
      6
      7 void Speculate(int[] sharePrice, int n) {
      8 /**
      9     @param   max is the gold price two weeks from today
   ****              ↑ max not in parameter list
     10     @return  to sanity after Wall Street crashes
   ****     ↑ void functions cannot return values
     11 */
     12 }
     13
     14 /** ParvaDoc comments cannot appear in arbitrary places between functions! */
   **** ↑ EOF expected
```

Producing a simple text file (say `Applic.txt`) from a program (say `Applic.pav`) that contains such documentation might be a useful first step. However, as mentioned earlier, a better system would produce an HTML version (say `Applic.htm`) with HTML tags inserted to allow a browser to display the documentation in a convenient form. This is not particularly difficult to do. If you go on to derive a system to generate this format you might like to study the simple web page supplied in the files kit for the examination, which exemplifies the sorts of HTML tags that might be incorporated into the output file.

Lastly, a system like ParvaDoc is easily enhanced to allow users to insert their own HTML tags into the documentation comments. For example, if the ParvaDoc comment above were extended to have clauses reading

```
int ReduceLength(int[] list, int n, int length) {
/** ...
    @see @<a href=sunday.htm>"Algorithms for a Special Sunday" (2005)@</a>
    @author PDT @<i>with a little help from my friends@</i>@<ul>
        @<li>John
        @<li>Paul
        @<li>George
        @<li>Ringo
    @</ul>
    ...
*/
```

a hyperlink to "`sunday.htm`" would appear in the web page in the "References:" section, and the names of the four friends would be rendered by the browser as a bulleted list in the "Author:" section. It is suggested that HTML tags can be introduced into commentary in this way with an `@` character immediately preceding the tag, in the same way that the `@` character is the first one in a tag like `@see`.

Hints: Do not alter the basic Parva grammar more than is absolutely necessary - do not be tempted to add the sorts of extensions that have formed part of the practical course this year. If you cannot develop a solution that correctly deals with all three kinds of comments, try to develop one assuming that the source program will only incorporate "ParvaDoc" comments. It is not required that you produce a solution that can derive both "simple text" and/or "html" text - develop a solution for one of these only.

Free Information

The following is an unattributed grammar for Parva, level 2 (which allows for simple multifunction programs)

```
COMPILER Parva
/* Parva level 2 grammar  - Coco/R for C# or Java
   P.D. Terry, Rhodes University, 2003
   Grammar only  -  LL(1) */

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
  = {   FuncOrVarDeclarations
      | ConstDeclarations
    } .

  FuncOrVarDeclarations
  = Type Ident ( Function | GlobalVars ) .

  Function
  = "(" FormalParameters ")" Block .

  GlobalVars
  = OneGlobal { "," Ident OneGlobal } ";" .

  OneGlobal
  = [ "=" Expression ] .

  FormalParameters
  = [ OneParam { "," OneParam } ] .

  OneParam
  = Type Ident .

  Block
  = "{" { Statement } "}" .

  Statement
  =   Block | AssignmentOrCall | ";"
    | ConstDeclarations | VarDeclarations
    | IfStatement | WhileStatement
    | HaltStatement | ReturnStatement
    | ReadStatement | WriteStatement .

  ConstDeclarations
  = "const" OneConst { "," OneConst } ";" .

  OneConst
  = Ident "=" Constant .

  Constant
  =   IntConst | CharConst
    | "true" | "false" | "null" .

  VarDeclarations
  = Type OneVar { "," OneVar } ";" .

  OneVar
  = Ident [ "=" Expression ] .
```

```
AssignmentOrCall
= Designator
  (   "=" Expression
    | "(" Arguments ")"
  ) ";" .

Designator
= Ident [ "[" Expression "]" ] .

Arguments
= [ OneArg { "," OneArg } ] .

OneArg
= Expression .

IfStatement
= "if" "(" Condition ")" Statement .

WhileStatement
= "while" "(" Condition ")" Statement .

ReturnStatement
= "return" [ Expression ] ";" .

HaltStatement
= "halt" ";" .

ReadStatement
= "read" "(" ReadElement { "," ReadElement } ")" ";" .

ReadElement
= StringConst | Designator .

WriteStatement
= "write" "(" WriteElement { "," WriteElement } ")" ";" .

WriteElement
= StringConst | Expression .

Condition
= Expression .

Expression
= AddExp [ RelOp AddExp ] .

AddExp
= [ "+" | "-" ] Term { AddOp Term } .

Term
= Factor { MulOp Factor } .

Factor
=   Designator [ "(" Arguments ")" ]
  | Constant
  | "new" BasicType "[" Expression "]"
  | "!" Factor | "(" Expression ")" .

Type
= "void" | BasicType [ "[]" ] .

BasicType
= "int" | "bool" .

AddOp
= "+" | "-" | "||" .

MulOp
= "*" | "/" | "%" | "&&" .

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
```