RHODES UNIVERSITY

November Examinations - 2007

Computer Science 301 - Paper 1

Examiners:
Prof P.D. Terry
Prof S. Berman

Time 3 hours
Marks 180
Pages 12 (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 Parva compiler that could handle aspects of a `string` *type. 16 hours before the examination a complete grammar and other support files for building such a system were supplied to students, along with an appeal to study this in depth (summarized in "Section D"). During the examination, candidates were given machine executable versions of the Coco/R compiler generator, the files needed to build the basic system, access to a computer, and machine readable copies of the questions.)*

Section A [100 marks]

A1. (*Compiler structure)* A syntax-directed compiler usually incorporates various components, of which the most important are the scanner, parser, constraint analyser, error reporter, code generator, symbol table handler and I/O routines. Draw a diagram indicating the dependence of these components on one another, and in particular the dependence of the central syntax analyser on the other components. Also indicate which components constitute the *front end* and which the *back end* of the compiler. [10 marks]

A2. *(Recursive descent parsers)* The following Cocol grammar describes the form of an index to a textbook and should be familiar from the practical course.

```
COMPILER Index $CN
/* Grammar describing index in a book
   P.D. Terry, Rhodes University, 2007 */

CHARACTERS
/* Notice the careful and unusual choice of character sets */
  digit      = "0123456789" .
  startword  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz('" + '"' .
  inword     = startword + digit + "-+)" .
  eol        = CHR(10) .

TOKENS
/* Notice the careful and unusual definition for word */
  word       = startword { inword } .
  number     = digit { digit } .
  EOL        = eol .

IGNORE CHR(0) .. CHR(9) + CHR(11) .. CHR(31)

PRODUCTIONS
  Index      = { Entry } EOF .
  Entry      = Key References EOL .
  Key        = word { "," word | word } .
  References = DirectRefs | CrossRef .
  DirectRefs = PageRefs { "," PageRefs  } .
  PageRefs   = [ "Appendix" ] number [ "-" number ] .
  CrossRef   = "--" "see" Key .
END Index .
```

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

```
EOFSym, noSym, EOLSym, wordSym, numberSym, appendixSym, commaSym,
dashSym, dashDashSym, seeSym
```

Develop a hand-crafted recursive descent parser for recognizing the index of a book based on the grammar above. *(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.)* [16 marks]

A3. *(Grammars)* 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 $A \in N$

(3) A context-free grammar

(4) A reduced grammar

(b) In terms of the notation here, concisely state the two rules that must be satisfied by the productions of a context free grammar in order for it to be classified as an LL(1) grammar. [6 marks]

(c) Describe the language generated by the following grammar, using English or simple mathematics. [2 marks]

$S \rightarrow A\,B$
$A \rightarrow a\,A \mid a$
$B \rightarrow b\,B\,c \mid bc$

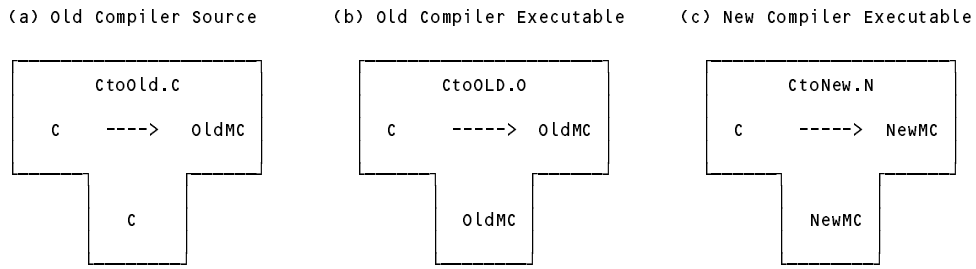
(d) Is the grammar in (c) an LL(1) grammar? If not, why not, and can you find an equivalent grammar that *is* LL(1)? [3 marks]

(e) The following grammar describes strings comprised of an equal number of the characters a and b, terminated by a period, such as *aababbba*. Is this an LL(1) grammar? Explain. [2 marks]

$S \rightarrow B\,.$
$B \rightarrow a\,B\,b\,B \mid b\,B\,a\,B \mid \varepsilon$

(f) "Keep it as simple as you can, but no simpler" said Einstein. Strings that might be members of the language of (e) can surely be accepted or rejected by a very simple algorithm. Give such an algorithm, using a Java-like notation. [3 marks]

A4. *(T diagrams)* The process of "porting" a compiler to a new computer incorporates a *retargetting* phase (modifying the compiler to produce target code for the new machine) and a *rehosting* phase (modifying the compiler to run on the new machine). Illustrate these two phases for porting a C compiler, by drawing a set of T diagrams. Assume that you have available the compilers (a) and (b) below and wish to produce compiler (c). [16 marks]



(You may conveniently make use of the outline T-diagrams at the end of this paper; complete these and attach the page to your answer book.)

A5. *(Attributed Grammars in Cocol)* XML (eXtensible Markup Language) is a powerful notation for marking up data documents in a portable way. XML code looks rather like HTML code, but has no predefined tags. Instead a user can create customized markup tags, similar to those shown in the following extract.

```
<!-- comment - a sample extract from an XML file -->
<personnel>
  <entry>
    <name>John Smith</name>
  </entry>
  <entry_2>
    <name>Joan Smith</name>
    <address/>
    <gender>female</gender>
  </entry_2>
</personnel>
```

An *element* within the document is introduced by an *opening tag* (like `<personnel>`) and terminated by a *closing tag* (like `</personnel>`), where the obvious correspondence in spelling is required. The name within a tag must start with a letter or lowline character (_), and may then incorporate letters, lowlines, digits, periods or hyphens before it is terminated with a > character. Between the opening and closing tags may appear a sequence of free format *text* (like `John Smith`) and further *elements* in arbitrary order. The free format text may not contain a < character - this is reserved for the beginning of a tag. An *empty element* - one that has no internal members - may be terminated by a closing tag, or may be denoted by an *empty tag* - an opening tag that is terminated by `/>` (as in `<address/>` in the above example). Comments may be introduced and terminated by the sequences `<!--` and `-->` respectively, but may not contain the pair of characters `--` internally (as exemplified above).

Develop a Cocol specification, incorporating suitable `CHARACTER` sets and `TOKEN` definitions for

(a) opening tags,
(b) closing tags,
(c) empty tags,
(d) free format text

and give `PRODUCTIONS` that describe complete documents like the one illustrated. You may do this conveniently on the page supplied at the end of the examination paper.

Tags must be properly matched. A document like the following must be rejected

```
<bad.Tag>
   This is valid internal text
   <okayTag>
     More internal stuff
   </okayTag>
</badTag>  <!-- badTag should have been written as bad.Tag -->
```

Show how your grammar should be attributed to perform such checks. [18 marks]

Incidentally, it should be noted that the full XML specification defines far more features than those considered here!

A6. *(Code generation)* A BYT (Bright Young Thing) has been nibbling away at writing extensions to her first Parva compiler. It has been suggested that a function that will return the maximum element from a variable number of arguments would be a desirable addition, one that might form part of an expression as in

```
a = max(x, y, z) + 5 - max(a, max(c, d));
```

and that this could be achieved by extending the production for a *Factor* in a fairly obvious way, and adding a suitable opcode to the PVM. To refresh your memory, the production for *Factor* in the simple Parva compiler is defined as follows:

```
Factor<out int type>                  (. int value = 0;
                                         type = Entry.noType;
                                         int size;
                                         DesType des;
                                         ConstRec con; .)
=    Designator<out des>              (. type = des.type;
                                         switch (des.entry.kind) {
                                           case Entry.Var:
                                             CodeGen.dereference();
                                             break;
                                           case Entry.Con:
                                             CodeGen.loadConstant(des.entry.value);
                                             break;
                                           default:
                                             SemError("wrong kind of identifier");
                                             break;
                                         } .)
   | Constant<out con>                (. type = con.type;
                                         CodeGen.loadConstant(con.value); .)
   | "new" BasicType<out type>        (. type++; .)
     "[" Expression<out size>         (. if (!isArith(size))
                                             SemError("array size must be integer");
                                         CodeGen.allocate(); .)
     "]"
   | "!" Factor<out type>             (. if (!isBool(type)) SemError("boolean operand needed");
                                         else CodeGen.negateBoolean();
                                         type = Entry.boolType; .)
   | "(" Expression<out type> ")"
.
```

while a sample of the opcodes in the PVM that deal with simple arithmetic and logic arithmetic are interpreted with code of the form

```
case PVM.ldc:           // push constant value
  push(next());
  break;
case PVM.add:           // integer addition
  tos = pop(); push(pop() + tos);
  break;
case PVM.sub:           // integer subtraction
  tos = pop(); push(pop() - tos);
  break;
case PVM.not:           // logical negation
  push(pop() == 0 ? 1 : 0);
  break;
```

Suggest, in as much detail as time will allow, how the *Factor* production and the interpreter would need to be changed to support this language extension.

Allow your `max` function to take one or more arguments, and ensure that it can only be applied to arithmetic arguments. Assume that a suitable `CodeGen` routine can be introduced to generate any new opcodes required. [16 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.

It was in 1988, at the second of a series of conferences held by a group charged with developing a rigorous standard for the definition of Modula-2, that one Susan Eisenbach made a telling observation: "Modula-2 would be the perfect language if we could add just one more feature. The difficulty is that nobody can decide what that one feature should be". She was right. Language designers cannot resist the temptation to add extension after extension to their brainchild.

So it is with Parva.

Until yesterday, the Parva language that you have got to know so well had but three basic data types - integer, boolean, and character. Although it made provision for the use of string constants in `read` and `write` statement lists:

```
void main() { $C+
  int i;
  read("Supply a value for i" , i);
  write("The value you supplied was ", i);
}
```

it did not provide for the declaration and use of string variables:

```
void main() { $C+
  string yourName, myName, theBoff;
  myName = "Pat";
  theBoff = myName;
  read("What is your name? ", yourName);
  write(myName, " is pleased to meet ", yourName);
  if (myName != yourName)
    writeLine(" (Our names are different)");
  if (upper(myName) == upper(yourName))
    writeLine("(Our uppercased names are the same)");
}
```

Yesterday you made history when you were invited to develop a version of the Parva compiler that at last incorporated a `string` type, one which should have been able to do the following at least:

(a) Declare variables and arrays of type `string`
(b) Assign constant strings to such variables, and values of string variables to other string variables
(c) Read and write values for strings
(d) Compare two strings for equality or inequality
(e) Provide a function for converting a string to UPPERCASE.
(f) Perform any necessary semantic and constraint checking on strings.

Later in the day you were provided with a sample solution to that challenge. Continue now to answer the following unseen questions.

B7. The compiler as provided does not allow you to declare strings as named constants. Suggest how this might be done, to allow, for example: [5 marks]

```
void main () {
  const title = "The thin edge of the wedge";
  string s = title;
}
```

B8. String handling libraries usually make provision for determining the length of a string. How would the compiler, code generator and interpreter have to be extended to allow for code like

```
void main () {
  string str;
  read(str);
  write(str, length(str) );
}
```

where the `length(str)` function can only be applied to a single parameter expression of `string` type. [10 marks]

B9. What does the system currently output if a program like the following is compiled and run

```
void main () {
  string str;  // declared
  write(str);  // but never properly defined
}
```

Explain your reasoning. [10 marks]

B10. The situation in B9 is actually symptomatic of a larger problem. Some implementations of languages ensure that all variables not explicitly "initialised" at the point of creation are implicitly initialised to a known value - say zero - anyway. How could your compiler be changed to incorporate this device (both for simple variables and also for arrays)? [15 marks]

B11. How could the system be changed to report situations like that in B9 as runtime errors? [5 marks]

B12. The compiler currently allows for string comparisons for equality and inequality. How would it need to be modified to allow for "ordering" comparisons, such as exemplified by [20 marks]

```
void main () {
  string s1, s2;  // declared
  read(s1, s2);   // initialised
  if (s1 < s2)    // compare
    write(s1, " is lexically less than ", s2);
}
```

B13. String concatenation (joining) is another useful feature. How would your compiler, code generator and interpreter have to be modified to allow for a program like [15 marks]

```
void main () {
  string s1, s2;  // declared
  read(s1, s2);   // initialised
  string joined = s1 + s2;
  write(joined);
}
```

Section C

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

Candidates were provided with the basic ideas, and were invited to extend a version of the Parva compiler to incorporate a `string` type.

It was pointed out that string types in languages that support them usually come with a complete support library of useful functions, but to begin with they might limit themselves to a system that would do the following:

(a) Declare variables and arrays of type `string`
(b) Assign constant strings to such variables, and values of string variables to other string variables
(c) Read and write values for strings
(d) Compare two strings for equality or inequality
(e) Provide a function for converting a string to UPPERCASE.

They were provided with an exam kit for Java or C#, containing a working compiler for integer and boolean types, along with a suite of simple, suggestive test programs. They were told that later in the day some further ideas and hints would be provided.

Section D

(Summary of free information made available to the students 16 hours before the formal examination.)

A complete Parva compiler, incorporating one approach to the basic provision of a simple `string` type, was supplied to candidates in a later version of the examination kit. They were encouraged to study it in depth and warned that questions in Section B would probe this understanding; few hints were given as to what to expect, other than that they might be called on to comment on the solution, and perhaps to make some modifications and extensions. They were also encouraaged to spend some time thinking how any other ideas they had during the earlier part of the day would need modification to fit in with the solution kit presented to them.

Free information

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
```

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. The modification for the generic version should be familiar.

```
import java.util.*;

class ArrayList
// Class for constructing a list of objects

  public ArrayList()          // generic:  public ArrayList<type> ()
  // 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 Object set(int index, Object o)
  // Stores an object o at 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
```

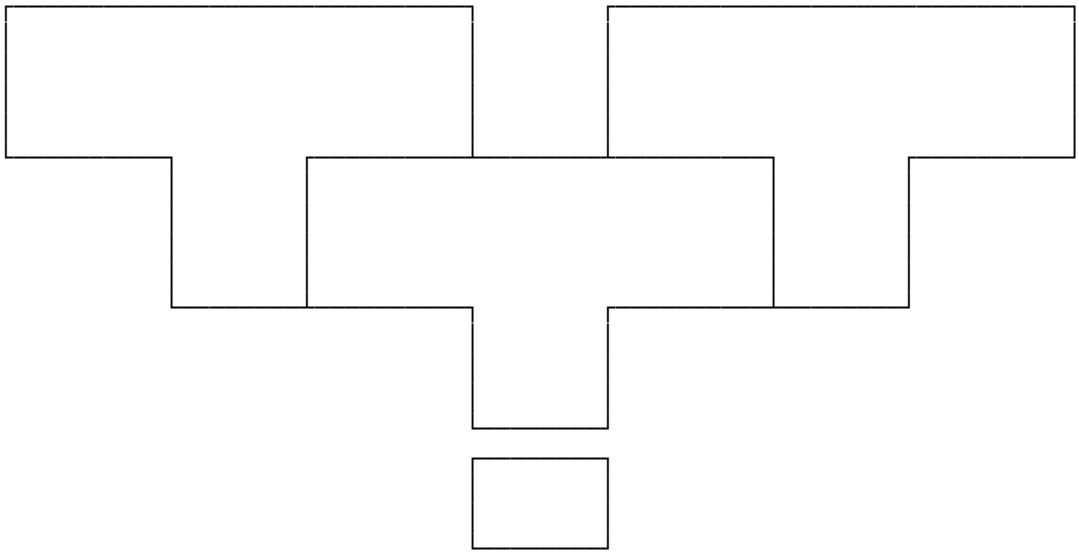
Simple list handling in C#

The following is the specification of useful members of a C# list handling class. The modification for the generic version should be familiar.

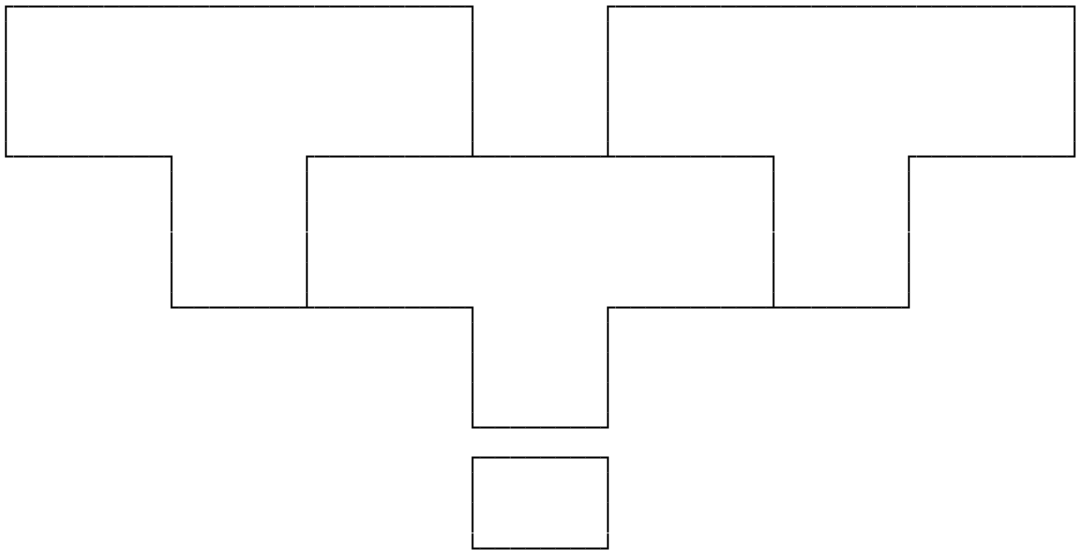
```
using System.Collections;

class ArrayList
// Class for constructing a list of objects

  public ArrayList()         // generic: public ArrayList<type> ()
  // Empty list constructor

  public int Add(object o)
  // Appends o to end of list

  public object this [int index] {set; get; }
  // Inserts or retrieves an object in position index
  // list[index] = object;  object = list[index]

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

  public int Count { get; }
  // Returns number of elements in list

  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 void Remove(object o)
  // Removes object o from list

  public void RemoveAt(int index)
  // Removes the object at position index

} // ArrayList
```

A4. T Diagrams for a compiler port Student number

Retargetting the compiler



Rehosting the compiler



A5. Cocol grammar for describing elements of an XML subset Student number

```
using Library;

COMPILER XML $CN
/* Parse a set of simple XML elements */

CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  incomment = ANY - "-" .









TOKENS
  opentag  =


  closetag =


  emptytag =


  text     =



PRAGMAS /* We cannot use the comment feature of Cocol which only allows
           two character delimiters */

  comment = "<!--" { incomment | '-' incomment } "-->" .

IGNORE  CHR(0) .. CHR(31)

PRODUCTIONS
   XML   =


END XML.
```