RHODES UNIVERSITY

November Examinations - 2011

Computer Science 301 - Paper 2 - Solutions

Examiners:
Prof P.D. Terry
Prof D.G. Kourie

Time 4 hours
Marks 180
Pages 17 (please check!)

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

A word of advice: The influential mathematician R.W. Hamming very aptly and succinctly professed that "the purpose of computing is insight, not numbers".

Several of the questions in this paper are designed to probe your insight - your depth of understanding of the important principles that you have studied in this course. If, as we hope, you have gained such insight, you should find that the answers to many questions take only a few lines of explanation. Please don't write long-winded answers - as Einstein put it "Keep it as simple as you can, but no simpler".

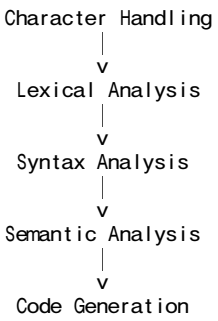
Good luck!

(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 devise a pretty-printer for Parva programs compatible with the Parva compiler/interpreter system studied in the course. Some 16 hours before the examination a complete grammar for a working pretty-printer and other support files for building this system were supplied to students, together with a grammar for the Mikra language, 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 systems, access to a computer, and machine readable copies of the questions.)

Section A: Conventional questions [102 marks]

QUESTION A1 **[5 + 6 + 4 = 15 marks]**

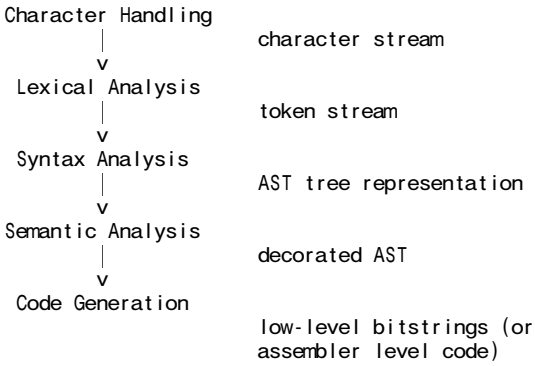
A compiler is often described as incorporating several "phases", as shown in the diagram below



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. [5 marks]

At the Character Handler level the program is essentially a sequence of characters. After Lexical Analysis it is effectively a stream of tokens. Syntax Analysis conceptually (or even in practice) builds a tree, and Semantic Analysis decorates the tree to give a form of intermediate code. Code Generation produces a file of low-level bitstrings, or possibly low-level assembler like code.



(b) Suggest - perhaps by example - the sort of errors that might be detected in each of the phases of the compilation process. [6 marks]

There is plenty of scope for variation in the answer to this question! At the character handling level the errors would arise from incomplete, missing, or corrupt files. At the lexical analysis stage one might fail to recognize a valid token at all (for example, if one came across an isolated \ in a Parva program, or had a comment that "opened" but did not "close"). At the syntax analysis stage errors such as improperly formed statements or missing punctuation might arise. Semantic analysis might throw up errors such as the use of undeclared identifiers or mismatched types in the operands of expressions. Code generation might lead to further file handling errors. For the sorts of "interpretive" compilers the class has seen, other code generation errors could arise from trying to generate too much code for the simulated memory of the virtual machine to handle.

(c) 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 above 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). [4 marks]

The insight called for here is that in Java one does not have to "declare before use" where methods are

concerned, which is very difficult to handle on a single pass without building an AST (which effectively allows for an easy second pass).

QUESTION A2 **[3 + 6 + 4 + 4 = 17 marks]**

(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]

divisibleBy4 = 0 | 1 (1 | 0) 0 0*

Most people do not notice that one needs a special case to handle "zero". "1 (1 | 0) 0 0" only caters for the non-zero values!*

(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. [6 marks]

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

non-zero = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit = 0 | non-zero
currency = R (0 | non-zero (digit | ε) (digit | ε) (, digit digit digit)) . digit digit*

```
CHARACTERS
   nonzero = "123456789" .
   digit   = nonzero + "0" .

TOKENS
   currency = "R" ( "0" | nonzero [ digit [ digit ]] { "," digit digit digit } )
              "." digit digit .
```

(c) 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% (respectively). Which of the phases of compilation identified in question A1 would this affect, which would it not affect, and why? [4 marks]

Essentially it affects only lexical analysis (for recognizing the alternative character strings permissible) and the small amount of semantic analysis needed to convert such strings to their corresponding integer values. In solutions submitted by candidates, a great many had clearly confused "types" and "values". Integer values can be represented in various ways - for example 123, 07FH, 0111% - but all these are values of the same type, so the grammar would not require the introduction of special compatiblity constraints, or the generation of special opcodes, for example. In a Cocol grammar for Parva the necessary changes would amount to the following - however, all this detail was not required.

```
CHARACTERS
  digit      = "0123456789" .
  hexDigit   = digit + "ABCDEF"
  binDigit   = "01"
  ...
TOKENS
  decNumber  = digit { digit } .
```

```
  hexNumber   = digit { hexdigit } "H" .
  binNumber   = bindigit { bindigit } "%" .
  ...

PRODUCTIONS
  ...

  IntConst<out int value>              (. int base = 10; .)
  = (   decNumber                      (. base = 10; .)
      | hexNumber                      (. base = 16; .)
      | binNumber                      (. base = 2; .)
    )                                  (. try {
                                            value = Integer.parseInt(token.val, base);
                                          } catch (NumberFormatException e) {
                                            value = 0; SemError("number too large"); }
                                       .) .
```

(d) A Cocol grammar for constructing a Parva compiler might contain the productions below for handling 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; .)              // version 1
  =                                    (. 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 has been made at the end of the `OneVar` production. A student has queried whether this production could be 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? [4 marks]

```
  OneVar<StackFrame frame, int type>   (. int expType; .)              // version 2
  =                                    (. 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); .)
     ]
  .
```

The insight looked for here may escape some readers. If one moves the "insert" action earlier, an initialization of the form

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

although problematic, will be acceptable - that is, would not generate any error messages, even though the "initialization" would be pretty meaningless. If the "insert" action is left until the end, an attempted initialization like the above would result in an "undeclared identifier" error message when the i was encountered as part of the Expression on the right of the = sign. Okay, so here is another meaningless initialization that would not be detected either:

```
int x, i = x + 1;
```

which is sad. Detecting the use of uninitialized variables is more complicated than it might at first appear!

Some candidates who get this question wrong might labour under the misconception that code like the above would store a value for i in the symbol table. Not so. However, a Parva ConstDeclaration, such as

```
const BattleOfTrafalgar = 1805;
```

would have stored 1805 in the symbol table. The two kinds of declarations are quite different.

QUESTION A3 **[2 + 3 + 2 + 4 + 10 + 3 = 26 marks]**

Long ago, the Romans represented 1 through 10 by the strings

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

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

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

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

An ambiguous grammar is one for which at least one acceptable sentence can be derived in two or more different ways.

(b) Explain carefully why this particular grammar is ambiguous? [3 marks]

Because the strings `VI` *and* `VII` *can be derived in two ways, depending on which of the optional* `I` *tokens is consumed when applying the production for* `StartV`

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

Two grammars are equivalent if they define exactly the same set of sentences, but use different production rules to do so.

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

```
COMPILER Roman

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

(e) Even though the grammar above is ambiguous, develop a matching hand-crafted recursive descent parser for it, similar to those that were developed in the practical course.

Assume that you have `accept` and `abort` routines like those you met in the practical course, and a scanner `getSym()` that can recognise tokens that might be described by an

```
EOFSym, noSym, commaSym, periodSym, iSym, vSym, xSym
```

Your parser should detect and report errors, but there is no need to incorporate "error recovery". [10 marks]

```
void Roman (void) {
// Roman = Number { "," Number } "." .
  Number();
  while (sym == commaSym) {
    GetSym(); Number();
  }
  accept(periodSym, ". expected");
}

void Number (void) {
// Number = StartI | StartV | StartX .
  switch (sym) {
    case ISym : StartI(); break;
    case VSym : StartV(); break;
    case XSym : StartX(); break;
    default: Abort("Unexpected symbol");
  }
}

void StartI (void) {
// StartI = "I" ( "V" | "X" | [ "I" [ "I" ] ] ) .
  accept(ISym, "I expected");
  switch (sym) {
    case VSym : GetSym(); break;
    case XSym : GetSym(); break;
    case ISym : GetSym(); if (sym == ISym) GetSym(); break;
    default: break;
  }
}

void StartV (void) {
// StartV = "V" [ "I" ] [ "I" ] [ "I" ] .
  accept(VSym, "V expected");
  if (sym == ISym) GetSym();
  if (sym == ISym) GetSym();
  if (sym == ISym) GetSym();
}

void StartX (void) {
// StartX = "X" .
  accept(XSym, "X expected");
}
```

(f) If the grammar is ambiguous (and thus cannot be LL(1)), does it follow that your parser might fail to recognize a correct sequence of Roman numbers, or might report success for an invalid sequence? Justify your answer. [3 marks]

It will work fine, as it will bind the `I` *tokens to the first occurrence of the option consuming code in the function* `StartV`.

QUESTION A4 **[3 + 3 = 6 marks]**

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

```
WhileStatement<StackFrame frame>      (. Label startLoop = new Label(known); .)
=  "while" "(" Condition ")"          (. Label loopExit = new Label(!known);
                                         CodeGen.branchFalse(loopExit); .)
   Statement<frame>                   (. CodeGen.branch(startLoop);
                                         loopExit.here(); .) .
```

This generates code that matches the template

```
startLoop:    Condition
              BZE  loopExit
              Statement
              BRN  startLoop
loopExit:
```

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

```
whileLabel:   BRN  testLabel
loopLabel:    Statement
testLabel:    Condition
              BNZ  loopLabel
loopExit:
```

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

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

While this might be the case on some pipelined architectures, on the interpreted system used in the course it would probably make little difference. Far more time is likely to be spent executing the many statements that make up the loop body than in executing an extra unconditional branch instruction.

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

It would be hard to implement using the simple minded code generator the students saw in this course, as we should need to delay generating the code for `Condition` *until after we had generated the code for a* `Block`*. Generating the various branch instructions would be easy enough, of course. In a more sophisticated compiler that built a tree representation of the program before emitting any code, judicious tree walking would allow one to get the effect relatively simply.*

QUESTION A5 **[6 + 10 + 4 = 20 marks]**

Scope and *Existence/Extent* are two terms that come up in any discussion of the implementation of block-structured languages.

(a) Briefly explain what these terms mean, and the difference between them. [6 marks]

This is all straightforward bookwork stuff: "Scope" is a compile time concept, relating to the areas of code in which a variable identifier can be recognised. "Existence" is a run time concept, relating to the time during which a variable has storage allocated to it. In a simple block structured language such as those studied on this course the concepts are tied together to the extent that the scope area of variables is bound to the static code of the block in which they have been declared, and at run time storage is usually allocated and deallocated to those variables as the blocks are activated and deactivated.

(b) Briefly describe a suitable mechanism that could be used for symbol table construction to handle scope rules and offset addressing for variables in a language like Parva. Illustrate your answer by giving a snapshot of the symbol table at each of the points indicated in the code below. (The first one has been done for you, and you can fill in the rest on the addendum page.) [10 marks]

The underlying mechanism is to use a stack for the symbol table, with markers indicating the points at which a new scope is encountered; when the scope dies the symbol table can be cut back to the corresponding marker. In the simple Parva implementation used in this course, variables could at compile time be given offset addresses that would, at run time, be subtracted from a run time "base pointer" address. Simplified, this becomes:

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

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

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

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

Point 1

Name	list	i	j	k							
Offset	0	1	2	3							

Point 2

Name	list	i	j	k	a	b	c				
Offset	0	1	2	3	4	5	6				

Point 3

Name	list	i	j	k	c	a	d				
Offset	0	1	2	3	7	8	9				

Point 4

Name	list	i	j	k	b	last					
Offset	1	6	7	8	10	11					

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

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

then the code would be acceptable to a C++ compiler but not to a Parva, Java or C# compiler, as C++ allows programmers much greater freedom in reusing/redeclaring identifiers in inner scopes. What benefits do you suppose language designers would claim for either greater or reduced freedom? [4 marks]

C++ proponents would probably claim that this gave programmers much greater freedom to choose local identifiers apposite to the block of code under consideration, without having perpetually to look over their shoulders at what identifiers had been declared already/elsewhere. Java proponents would claim that inventing a few more names would not be a big deal, and would claim that protection from accidentally masking out global identifiers that they might otherwise want to reference would be a useful form of protection.

QUESTION A6 **[8 marks]**

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

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

Discuss the extent to which these "errors" might be detected by suitable extensions of the Parva compiler/interpreter system developed in this course. You do not need to give the algorithms in detail, but you might like to structure your answer on the following lines: [8 marks]

Variables declared but never referenced:

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

Variables referenced before their values are defined:

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

Variables declared but never referenced:

Detectable at compile time by marking their symbol table entries "unreferenced" at the point of declaration, marking them "referenced" if they are ever used, and then checking when the scope is closed to find those still marked "unreferenced".

Variables referenced before their values are defined:

Not detectable at compile time (even with extensive data flow analysis) because it is impossible to cover all the possible paths through the algorithm.

Detectable at run time time (especially in an interpretive system) by marking all variables "undefined" in some way as storage is allocated to them and generating code that checks each time a variable is referenced (loaded into a register) that it has been marked "defined", and code that marks each variable as "defined" if a value is ever stored at the appropriate address. This can be done in an interpretive system by defining the "memory" as an array of structs, with one value field and one boolean field. It can be done more crudely and less reliably by using some highly unlikely value such as MAXINT to act as an "undefined" value.

QUESTION A7 **[10 marks]**

Here is a true story. A few years ago I received an e-mail from one of the many users of Coco/R in the world. He was looking for advice on how best to write Cocol productions that would describe a situation in which one non-terminal `A` could derive four other non-terminals `W, X, Y, Z`. These could appear in any order in the sentential form, but there was a restriction that each one of the four had to appear exactly once. He had realised that he could enumerate all 24 possibilities, on the lines of

```
A = W X Y Z | W X Z Y | W Y X Z | ....
```

but observed astutely that this was tedious. Furthermore, it would become extremely tedious if one were to be faced with a more general situation in which one non-terminal could derive *N* alternatives, which could appear in any order, subject to the restriction that each should appear exactly once.

Write the appropriate parts of a better Cocol specification that describes the situation and checks that the restrictions are correctly met. (Restrict your answer to the case of 4 derived non-terminals, as above.) [10 marks]

It's the action that counts! Simply relax the syntax to allow any number of W, X, Y, Z to appear in any order; count how many of each there are, and then check afterwards that each has appeared exactly once:

```
A =                 (. int W = 0, X = 0, Y = 0, Z = 0; .)
  {   W             (. W++; .)
    | X             (. X++; .)
    | Y             (. Y++; .)
    | Z             (. Z++; .)
  }                 (. if (W * X * Y * Z != 1) SemError("invalid list"); .) .
```

Sadly, few people usually come up with this (preferring complicated solutions that used actions to build up tables and so on. A few will come up with solutions which try to record strings, but note that the problem was posed in terms of non terminals W, X, Y and Z, so that `token.val` *would have been of no use really. Ah well. I keep trying to tell people that behind every problem there is a simple solution waiting to be discovered. There really is!*

Section B: Converting Mikra programs to Parva [78 marks]

Please note that there is no obligation to produce machine readable solutions 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 EXAMC.ZIP or EXAMJ.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

Yesterday you made history when you were invited to produce a pretty-printer system for Parva programs.

Later in the day you were provided with a sample solution to that challenge, and the files needed to build that system have been provided to you again today. You were also provided with an unattributed grammar for Mikra, another small language.

Today we have gone one step further and provided you not only with these systems, but also with all the files needed to produce a pretty-printer for Mikra programs.

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 2011 is proving to be no exception. Believe it or not, the Department is due to demonstrate the use of Mikra this afternoon, but have managed to lose every single copy of the Mikra compilers once so popular among students!

What should one do in such an emergency? Dial 911? Dial the Emeritus Professor?

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? You can construct a system that will pretty-print Mikra programs. Simply modify this so that the pretty-printer produces Parva output in place of Mikra output, and then the Department will never notice the difference - they will automagically be able to convert their Mikra programs to Parva and then compile them with the Parva compiler instead."

As usual, he's right! Mikra programs are really very similar to Parva ones, except for a little "syntactic sugar", as the following examples will illustrate:

```
program Entrance;
  begin
    writeLn("Hello world")
  end Entrance.
```

```
void main() {
  writeLine("Hello world");
} // main
```

```
program Exit;
  begin
    writeLn("Goodbye cruel world");
    writeLn("I'm off to join the circus")
  end Exit.
```

```
void main() {
  writeLine("Goodbye cruel world");
  writeLine("I'm off to join the circus");
} // main
```

```
program TimesTable;
  const limit = 12;
  var i, n : int;
  begin
    read("Which times table?", n);
    i := 1;
    while i <= limit do
      writeLn(i, n * i);
      inc(i)
    end (* while *)
  end TimesTable.
```

```
void main() {
  const limit = 12;
  int i, n;
  read("Which times table?", n);
  i = 1;
  while (i <= limit) {
    writeLn(i, n * i);
    i++;
  } /* while */
} // main
```

Go on now to answer the following questions explaining the details of the process. It is unlikely that you can complete a full system in the time available, so limit yourself to the modifications asked. As usual, a suite of simple test programs has been supplied for your use, suitably arranged to allow an incremental approach to be

followed.

As a hint, many of these refinements only require you to add or modify a few lines of the attributed grammar you have been given. Your answers should be given by showing the actual code you need, not by giving a long-winded description in English! Finally, not only have you been given a `PrettyMikra.atg` file, you have also been given a `Mikra2Parva.atg` file that aleady incorporates a few of the modifications.

QUESTION B8 **[3 marks]**

How would you classify a piece of software like `Mikra2Parva` - for example, is it a compiler, an assembler, a decompiler, an interpreter, some other name … Explain.

It is best described as a "high level compiler", because it translates from one high-level language to another one, rather than to low level object code. Students have seen another of these - X2C which translated Modula-2 to C.

QUESTION B9 **[5 marks]**

If Parva and Mikra are so similar, why is the grammar for one of them non-LL(1), while the grammar for the other one *is* LL(1)? Do you think this is significant? Explain briefly.

The grammar for Mikra is LL(1); that for Parva is not LL(1). The Parva grammar displays the famiiar "dangling else" phenomenon. The grammar for Mikra avoids this by requiring that statements like WhiileStatement, LoopStatement, ForStatement and in particular IfStatement are terminated by an explicit "end".

QUESTION B10 **[6 + 8 + 5 + 5 = 24 marks]**

A few of the productions for `Mikra2Parva` have already been converted in the grammar supplied to you, so that you can get an idea of how to complete many of the others. For example, you will find

```
PRODUCTIONS
  Mikra2Parva               (. String name; .)
  = "program"               (. CodeGen.append("void main() {"); .)
     Ident<out name>
     ";"                    (. CodeGen.indentNewLine(); .)
     Block
     Ident<out name>
     "."
  .

  Block
  = {   ConstDeclarations
      | VarDeclarations }
    "begin"
    StatementSequence
    "end"                   (. CodeGen.append(" // main"); .)
  .

  StatementSequence
  = Statement { ";"         (. CodeGen.newLine(); .)
    Statement }             (. CodeGen.exdentNewLine(); CodeGen.append("}"); .)
  .

  WhileStatement
  = "while"                 (. CodeGen.append("while ("); .)
    Condition "do"          (. CodeGen.append(") {"); CodeGen.indentNewLine(); .)
    StatementSequence
    "end"
  .

  HaltStatement
  = "halt"                  (. CodeGen.append("halt;"); .)
  .

  AssignOp
  = ":="                    (. CodeGen.append(" = "); .)
  .
```

which means that if you build the system and restrict the form of the Mikra programs to very simple ones like

```
program Stop;
  begin
    halt
  end Stop.
```

```
void main() {
  halt;
} // main
```

you should be able to convert them immediately. A few more changes on the same lines and you should be able to handle examples like

```
program DeepPhilosphy;
  begin
    write("I maintain that ");
    write((3 + 4) <> 6, " is true");
    writeLn
  end DeepPhilosophy+.
```

```
void main() {
  write("I maintain that ");
  write((3 + 4) != 6, " is true");
  writeLine();
} // main
```

(a) Make the appropriate changes to handle the differences in the syntax for arithmetic, boolean and assignment operators. [6 marks]

```
    NotOp
*   = "not"                 (. CodeGen.append("!"); .) .

    MulOp
    =   "*"                 (. CodeGen.append(" * "); .)
      | "/"                 (. CodeGen.append(" / "); .)
*     | "mod"               (. CodeGen.append(" % "); .)
*     | "and"               (. CodeGen.append(" && "); .) .

    AddOp
    =   "+"                 (. CodeGen.append(" + "); .)
      | "-"                 (. CodeGen.append(" - "); .)
*     | "or"                (. CodeGen.append(" || "); .) .

    RelOp
*   =   "="                 (. CodeGen.append(" == "); .)
*     | "<>"                (. CodeGen.append(" != "); .)
      | "<"                 (. CodeGen.append(" < "); .)
      | "<="                (. CodeGen.append(" <= "); .)
      | ">"                 (. CodeGen.append(" > "); .)
      | ">="                (. CodeGen.append(" >= "); .) .

    AssignOp
*   = ":="                  (. CodeGen.append(" = "); .) .
```

(b) Make the appropriate changes to handle the differences in the syntax for *read* and *write* statements. [8 marks]

```
    ReadStatement
    = (   "read"            (. CodeGen.append("read"); .)
          ReadList
*       | "readLn"          (. CodeGen.append("readLine"); .)
          (   ReadList
*           |               (. CodeGen.append("()"); .)
          )
*     )                     (. CodeGen.append(";"); .)
    .

    WriteStatement
    = (   "write"           (. CodeGen.append("write"); .)
          WriteList
*       | "writeLn"         (. CodeGen.append("writeLine"); .)
          (   WriteList
*           |               (. CodeGen.append("()"); .)
          )
*     )                     (. CodeGen.append(";"); .) .
```

(c) Mikra has a *repeat* loop where Parva has a *do-while* loop. How do you modify the system to transform a *repeat* loop to a *do-while* loop? [5 marks]

```
    RepeatStatement
*   = "repeat"                  (. CodeGen.append("do {"); CodeGen.indentNewLine(); .)
      StatementSequence
*     "until"                   (. CodeGen.append(" while (!("); .)
*     Condition                 (. CodeGen.append("));"); .) .
```

(d) Mikra has a *loop … exit … end* loop which has no direct equivalent in Parva, but is still easily handled. Make the appropriate changes to handle this construct. [5 marks]

```
    LoopStatement
*   = "loop"                    (. CodeGen.append("while (true) {"); CodeGen.indentNewLine(); .)
      StatementSequence
      "end" .

    ExitStatement
*   = "exit"                    (. CodeGen.append("break;"); .) .
```

QUESTION B11 **[5 marks]**

If Mikra can define a *repeat* loop by a production equivalent to

RepeatStatement = "repeat" Statement { ";" Statement } "until" Condition .

could the designer of Parva have defined a *do-while* loop by a production equivalent to

DoWhileStatement = "do" Statement { Statement } "while" "(" Condition ")" ";" .

If not, why not, and if so, why do you suppose Parva did not define it that way?

One cannot do this. The Parva grammar would become badly non-LL(1), as the token for the "closing" bracket in a DoWhileStatement would be "while", which could also be the first token in a WhileStatement, which you might wish to nest inside the DoWhileStatement..

QUESTION B12 **[5 marks]**

Make the appropriate changes to handle the differences in the syntax for casting operations.

```
    Factor                      (. String typeName; .)
    =   Designator | Constant
      | "new"                   (. CodeGen.append("new "); .)
        BasicType "["           (. CodeGen.append("["); .)
        Expression "]"          (. CodeGen.append("]"); .)
*     | "char"                  (. CodeGen.append("(char)"); .)
        "("                     (. CodeGen.append("("); .)
        Expression
        ")"                     (. CodeGen.append(")"); .)
*     | "int"                   (. CodeGen.append("(int)"); .)
        "("                     (. CodeGen.append("("); .)
        Expression
        ")"                     (. CodeGen.append(")"); .)
      | "("                     (. CodeGen.append("("); .)
        Expression
        ")"                     (. CodeGen.append(")"); .)
      | NotOp Factor .
```

QUESTION B13 **[12 marks]**

Make the appropriate changes to handle the differences in the way in which variables are declared.

```
    VarDeclarations
    = "var"
        VarList ";"             (. CodeGen.newLine(); .)
      { VarList ";"             (. CodeGen.newLine(); .)
      } .
```

```
    VarList                  (. String name;
*                               ArrayList<String> nameList = new ArrayList<String>(); .)
*   = Ident<out name>        (. nameList.add(name); .)
      { WEAK ","
*     Ident<out name>        (. nameList.add(name); .)
      } ":"
*     Type                   (. CodeGen.indentNewLine();
*                               for (int i = 0; i < nameList.size() - 1; i++) {
*                                 CodeGen.append(nameList.get(i));
*                                 CodeGen.append(", ");
*                               }
*                               CodeGen.append(nameList.get(nameList.size() - 1));
*                               CodeGen.append(";"); CodeGen.exdentNewLine(); .) .

*   Type                     (. boolean isArray = false; .)
*   = [ "array" "of"         (. isArray = true; .)
*     ] BasicType            (. if (isArray) CodeGen.append("[]"); .) .
```

After completing Questions B10 through B13 you will be able to handle quite a variety of programs. Do not bother to make further changes - they can wait for a later day!

QUESTION B14 **[5 marks]**

Do you suppose it is significant that the system will simply discard comments? Explain your reasoning, but do not bother to try to retain comments - programs without comments are found all too often in any case!

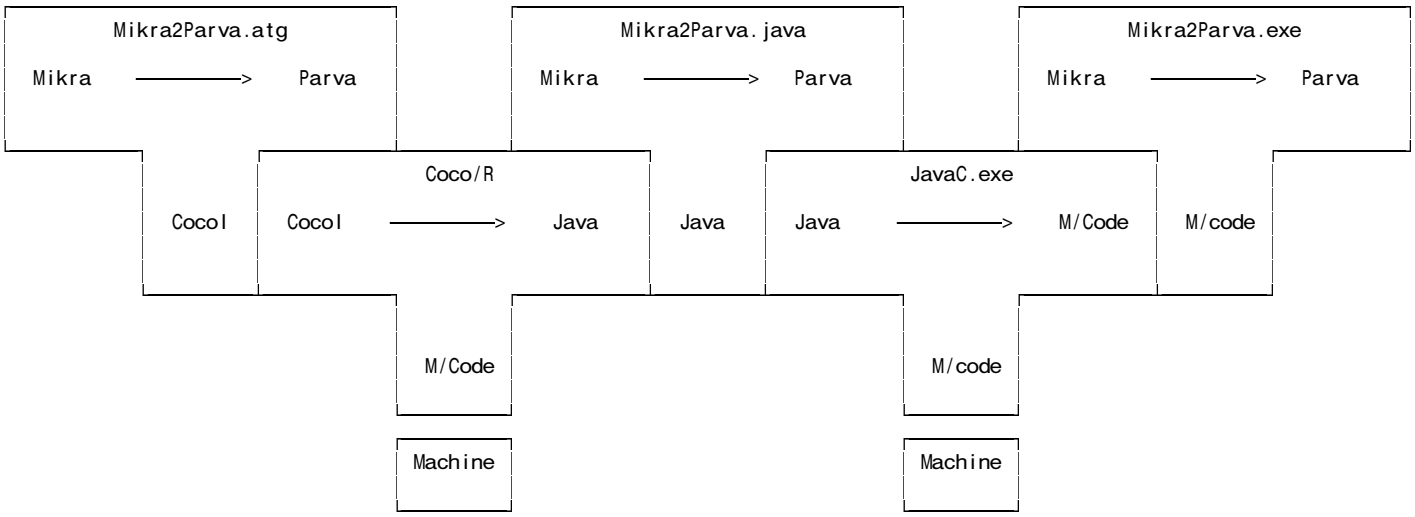
It is quite insignificant. Comments would be discarded by the Parva compiler, so if they been removed by Mikra2Parva before the Parva compiler sees the translated program it matters not a jot!

QUESTION B15 **[10 marks]**

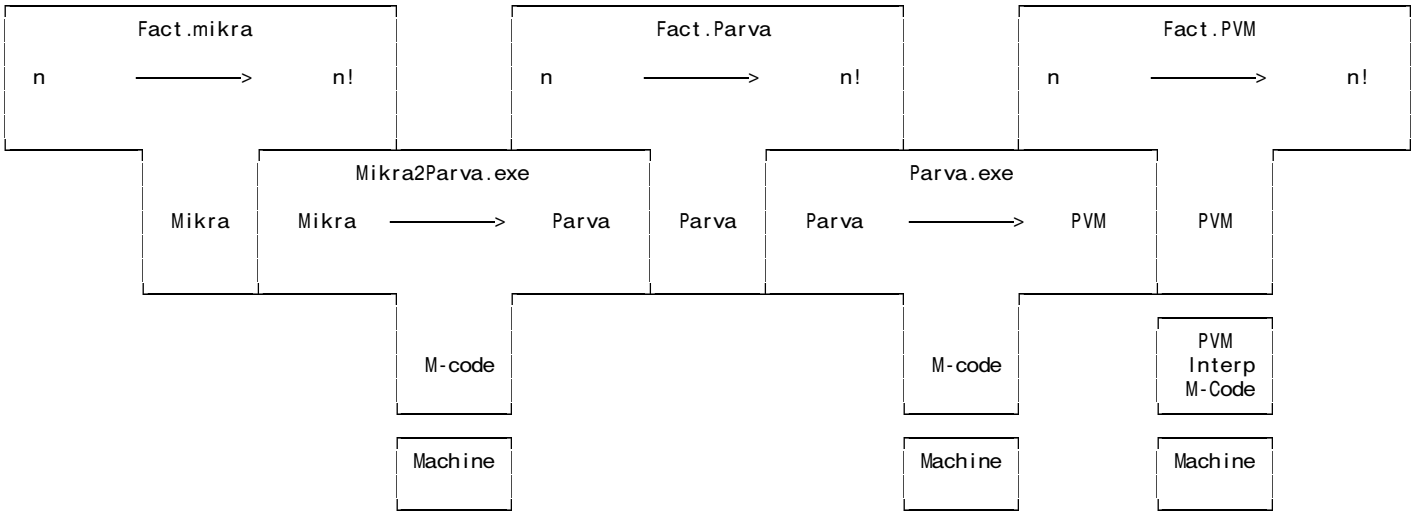
On the addendum page, complete the T-diagrams to denote

(a) the process of creating the `Mikra2Parva` executable;
(b) the process by which it is used to convert and then compile a Mikra program.

Hand this page in with your answer book.



Using the Mikra2Parva system



QUESTION B16 **[9 marks]**

It should not have escaped your attention that these pretty-printers have made no use of symbol tables or of semantic analysis. While this simplifies their construction, it might have implications for their use. Discuss this point in a little detail.

If the Mikra source program is syntactically and statically semantically correct, it should pass through the Mikra2Parva system very easily, and then the resulting Parva code should be acceptable to the Parva compiler. If the Mikra program is semantically incorrect - for example has undeclared variables or type mismatches - these will only be picked up by the Parva compiler. If the Mikra program is syntactically incorrect, such errors should be picked up by the Mikra2Parva system. This makes for easy development of Mikra2Parva, as should by now be obvious, but a user might become frustrated that errors are detected at different stages of what should be a seamless process.

The situation could be improved in two ways - build in the semantic checking into the Mikra2Parva system (essentially using the same sort of code as In the "front end" described in Chapter 13 of the text) or, secondly, by simply modifying the source of the full Parva compiler to accept the new Mikra syntax. As you will realise, this later process would be quite easy!