RHODES UNIVERSITY

November Examinations - 1998

Computer Science 301 - Paper 2

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

Time 3 hours
Marks 180

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. During the examination, candidates were given machine executable versions of the Coco/R compiler generator, and access to a computer.)

Section A [110 marks]

A1 Write a sentence or two to explain your understanding of the differences between the following pairs of concepts (keep the answers short and to the point; we do not need "essays").

(a) Nonterminals and terminals
(b) The front end and the back end of a translator
(c) A sentential form and a sentence
(d) Static semantics and dynamic semantics

[12]

A2 Write notes to show that you understand what is commonly meant by each of the terms *compiler, interpreter,* and *interpretive compiler*. Also indicate situations in which the use of each of these might be particularly favoured over the use of the other two.

[9]

A3 "Computer languages are not totally context-free, although syntactically they can almost always be described by context-free grammars". Give two examples of context-sensitive features of an imperative language with which you are familiar. If these features cannot be described by context-free grammars, how are they handled in practical syntax-directed compilers that are based on context-free grammars?

[6]

A4 Write down a regular expression (or a set of regular expressions) that define all possible words that have each vowel appearing exactly once, and in the correct order (vowels are a, e, i, o and u; an example of such a word is "facetious").

[3]

A5 The following forms part of a hand-crafted scanner for the language Clang whose familiar context-free grammar is supplied as an appendix to this paper

(A Pascal equivalent scanner also appears in the appendix; all these files are available in machine readable form in the exam kit).

In Clang, keywords like `BEGIN` and `WHILE` have to be distinguished from identifiers. Suggest the sort of modifications that would be needed to the scanner below to enable this to be done. (Your answer can be expressed either in C++ or in Pascal, and does not need to be syntactically perfect.)

[5]

```
void getSym(symbols &SYM)
{ int length = 0;                // index into SYM.Name
  while (ch <= ' ') getChar();  // Ignore spaces between tokens
  if (isalpha(ch))
  { SYM.sym = identsym;
    while (isalnum(ch)) { SYM.name[length++] = toupper(ch); getChar(); }
    SYM.name[length] = '\0';   // Terminate string properly
  }
  else if (isdigit(ch))        // Numeric literal
  { SYM.sym = numsym;
    while (isdigit(ch)) { SYM.name[length++] = toupper(ch); getChar(); }
    SYM.name[length] = '\0';   // Terminate string properly
  }
  else switch (ch) {
    case '<':
      getChar();
      if (ch == '=') { SYM.sym = leqsym; strcpy(SYM.name, "<="); getChar(); }
      else if (ch == '>') { SYM.sym = neqsym; strcpy(SYM.name, "<>"); getChar(); }
      else SYM.sym := lsssym; SYM.name := "<"
      break;

    ...
  }
}
```

A6 (a) Give a precise definition of the terms FIRST(A) and FOLLOW(A) as they are applied to grammar analysis.

(b) By computing relevant FIRST and FOLLOW sets, discuss whether the following grammar adheres to the LL(1) restrictions on parsing.

$N = \{ A , B , C , D , E \}$
$T = \{ w , x , y , z \}$
$S = A$
$P = A \Rightarrow BC \mid DE$
$B \Rightarrow [E]$
$C \Rightarrow x E$
$D \Rightarrow E \{ A \} (E \mid w)$
$E \Rightarrow y \mid z$

[18]

A7 Given the following set of productions for describing numerical expressions:

exp = *term* "↑" *exp* | *term* .
term = *term* "*" *factor* | *term* "/" *factor* | *factor* .
factor = *number* .

(a) Write a leftmost derivation for the string 2 ↑ 3 * 4 / 6.
(b) Draw a parse tree for this string.
(c) If the operators ↑ * and / represent the operations of raising to a power, integer multiplication, and integer (truncated) division respectively, what would be the value represented by this string?
(d) Would the value represented by the string 2 ↑ 2 ↑ 3 be 2^8 or 4^3? Why?
(e) How would you rewrite the grammar so that the string 2 ↑ 2 ↑ 3 would have the other meaning from that in (d)?

[15]

A8 Consider the Clang grammar in the appendix, and in particular the productions

```
OneConst   = identifier "=" number ";" .
Assignment = Variable ":=" Expression .
Variable   = Designator .
Expression = ( "+" Term | "-" Term | Term ) { AddOp Term } .
Term       = Factor { MulOp Factor } .
Factor     = Designator | number | "(" Expression ")" .
AddOp      = "+" | "-" .
MulOp      = "*" | "/" .
RelOp      = "=" | "<>" | "<" | "<=" | ">" | ">=" .
```

Suppose that we wished to extend the language so that we could write

```
OneConst   = identifier "=" ConstExpression ";" .
ConstExpression = Expression .
```

that is, we wish to be able to define constants in terms of expressions, rather than being restricted to positive integers. To do this we should need to write an attributed version of the grammar so that the parser could distinguish between constant expressions, which would be semantically acceptable in handling `OneConst`, and general expressions, which would be acceptable in handling `Assignment`.

(a) Making the simplifying assumption that an `Expression` in which only numbers appear as operands is a constant expression, but that the presence of a general `Designator` would not be allowed in a constant expression, suggest how one might attribute the productions for `Expression`, `Term` and `Factor` so that the `ConstExpression` parser could make the appropriate semantic check.

(b) Discuss clearly whether your solution has made use of *synthesized attributes* or *inherited attributes*.

(c) To develop a compiler one would, of course, have to add further to the grammar to incorporate calls to suitable code generating routines, and in this course we have discussed how this could be done. Would the routines that we illustrated - augmented only by the modifications suggested in (a) above - enable us to incorporate constant expressions into constant declarations, and if not, why not? (You do not have to show the development of the code generating routines in any detail, simply discuss the principles involved.)

(The Clang grammar is available to you in machine readable form; you are under no obligation to produce a machine readable solution, however.)

[20]

A9 After careful consideration of the overwhelming success of the Clang language, its inventor has decided to release Clang-2. In this language, the productions for `IfStatement` and `WhileStatement` are to be changed to

```
IfStatement     = "IF" Condition
                      "THEN" Statement { ";" Statement }
                    [ "ELSE" Statement { ";" Statement } ]
                  "END" .
WhileStatement  = "WHILE" Condition "DO"
                      Statement { ";" Statement }
                  "END" .
```

The proud inventor was overheard to say that this would dispel of the famous ″dangling else″ problem.

(a) What do you understand by the ″dangling else″ problem?
(b) Analyse in some detail whether the inventor′s claim is correct.

[10]

A10 Howls of protest arose when Clang-2 was released. All those millions of lines of code produced in the Braae Laboratory by generations of students were, at a stroke, rendered syntactically incorrect.

The inventor was unperturbed, and promised to use Coco/R to write a quick Clang to Clang-2 translator. He argued that all he needed to do was to write a sort of pretty printer that would reformat existing Clang programs with extra `END`s inserted into statements (and/or redundant `BEGIN`s removed) at appropriate places, and sketched out a few T-diagrams to show how he would do this. What did the T diagrams look like? (Make the assumption that you have available the executable versions of the Coco/R compiler-generator for either Pascal or C++, an executable version of a Pascal or C++ compiler, and at least one Clang program to be converted.)

[12]

```
          PROCEDURE GetSYM (VAR SYM : SYMBOLS);
            VAR
              Length : INTEGER  (* index into SYM.Name *);
            BEGIN
              IF NOT Started THEN BEGIN GetChar; Started := TRUE END;
              WHILE CH <= ' ' DO GetChar (* ignore spaces between tokens *);
              Length := 1;
              CASE CH OF
                'A' .. 'Z', 'a' .. 'z' :
                  BEGIN
                    SYM.Sym := IdentSym;
                    WHILE IsAlphaNum(CH) DO BEGIN
                      SYM.Name[Length] := UpCase(CH); INC(Length); GetChar
                    END;
                    SYM.Name[0] := CHR(Length - 1)  (* set length byte *)
                  END;
                '0' .. '9' :
                  BEGIN
                    SYM.Sym := NumSym;
                    WHILE IsDigit(CH) DO BEGIN
                      SYM.Name[Length] := UpCase(CH); INC(Length); GetChar
                    END;
                    SYM.Name[0] := CHR(Length - 1)  (* set length byte *)
                  END;
                '<' :
                  BEGIN
                    GetChar;
                    IF CH = '='
                      THEN BEGIN SYM.Sym := LeqSym; SYM.Name := '<='; GetChar END
                      ELSE IF CH = '>'
                        THEN BEGIN SYM.Sym := NeqSym; SYM.Name := '<>'; GetChar END
                        ELSE SYM.Sym := LssSym; SYM.Name := "<"
                  END;
                ...
              END
            END;

COMPILER Clang

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "(*" TO "*)"

CHARACTERS
  cr         = CHR(13) .
  lf         = CHR(10) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  instring   = ANY - "'" - cr - lf .
TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  string     = "'" (instring | "''") { instring | "''" } "'" .
PRODUCTIONS
  Clang             = "PROGRAM" identifier ";" Block "." .
  Block             = { ConstDeclarations | VarDeclarations } CompoundStatement .
  ConstDeclarations = "CONST" OneConst { OneConst } .
  OneConst          = identifier "=" number ";" .
  VarDeclarations   = "VAR" OneVar { "," OneVar } ";" .
  OneVar            = identifier [ UpperBound ] .
  UpperBound        = "[" number "]" .
  CompoundStatement = "BEGIN" Statement { ";" Statement } "END" .
  Statement         = [   CompoundStatement | Assignment     | IfStatement
                        | WhileStatement    | ReadStatement  | WriteStatement ] .
  Assignment        = Variable ":=" Expression .
  Variable          = Designator .
  Designator        = identifier [ "[" Expression "]" ] .
  IfStatement       = "IF" Condition "THEN" Statement .
  WhileStatement    = "WHILE" Condition "DO" Statement .
  Condition         = Expression RelOp Expression .
  ReadStatement     = "READ" "(" Variable { "," Variable } ")" .
  WriteStatement    = "WRITE"
                      [ "(" WriteElement { "," WriteElement }  ")" ] .
  WriteElement      = string | Expression .
  Expression        = ( "+" Term | "-" Term | Term ) { AddOp Term } .
  Term              = Factor { MulOp Factor } .
  Factor            = Designator | number | "(" Expression ")" .
  AddOp             = "+" | "-" .
  MulOp             = "*" | "/" .
  RelOp             = "=" | "<>" | "<" | "<=" | ">" | ">=" .
END Clang.
```

Section B [70 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 EXAMP.ZIP (Pascal) or EXAMC.ZIP (C++), modify any files that you like, and then copy all the files back to the blank diskette that will be provided.

The manufacturers of a microprocessor that is currently being developed have been provided with an assembler for it, but are keen to work at a higher level, and have called upon various software houses to bid for a contract to provide a simple compiler that will generate assembler code from a simple source language, which is itself in the process of being developed.

Potential suppliers have been asked to meet an initial specification that calls for a compiler to support a language that can declare simple variables, and have simple assignment statements, if-then-else statements and compound statements. Because things are not yet stable, they would like to be able to mix inline assembler code with higher level source code, so as to compensate for the absence of some high level features.

The microprocessor design is also rather unstable. At this stage all that is known of the instruction set is that it has five opcodes `MOV, CMP, BRN, BZE` and `BNZ`. These can be described as follows:

The load/store instructions are of the form

```
MOV  Rx, Ry     ;    Rx := Ry
MOV  Rx, a      ;    Rx := a
MOV  [a], Rx    ;    Mem[a] := Rx
MOV  [a], b     ;    Mem[a] := b
MOV  [a], [b]   ;    Mem[a] := Mem[b]
```

where `Rx` and `Ry` are chosen from `R1, R2, R3, R4, R5`, the machine's five registers, and `a` and `b` are positive integers.

The comparison instructions are of the form

```
CMP  x, y       ;    CPU.Z := (x = y)
CMP  x, [b]     ;    CPU.Z := (x = Mem[b])
CMP  [b], y     ;    CPU.Z := (Mem[b] = y)
```

where `x` and `y` can be any of `R1, R2, R3, R4, R5` or a simple positive integer, and `b` is a positive integer.

The absolute branch instructions are of the form

```
BRN  x          ;    CPU.PC := x
BZE  x          ;    if CPU.Z then CPU.PC := x
BNZ  x          ;    if not CPU.Z then CPU.PC := x
```

where `x` is a positive integer.

In the high-level language, variable names can be freely chosen, and no restriction is imposed against accidentally using a register name or opcode mnemonic as a variable name.

Just to make things interesting, the company has requested that the compiler not be case sensitive, and that variables can be tied, at the point of declaration to a specific machine register. Variables that are not tied in this way are sequentially assigned addresses in memory, decreasing from location 255.

Within the inline assembler code, however, while the opcode names are "reserved", other variable names may be used as operands.

All this has rather perplexed the host of firms who are competing for the lucrative contract, so the company has released a few sample programs, showing all these features in programs otherwise devoid of any real semantic interest! One of these is as follows (others will be found in the exam kit):

```
program example;
(* simple example of mixed language program *)
  int
    x, r4, r5,       (* some with register names *)
  bool
    CMP, MOV, c, d;  (* some with opcode names *)
  int
    r3 = r3,         (* r3 is attached to to register R3 *)
    register1 = r1,  (* register1 is attached to to register R3 *)

  begin
    x := r4;
    if x = r4
      then
        begin CMP := MOV; x := c; r3 := c; c := 203 end
      else
        if x = r5
          then MOV := c
          else MOV := r3;
    r3 := register1;
    if c = d
      then begin
        d := CMP;
        asm
          mov r1, r2;
          mov r4, 34;
          mov [34], 34;
          mov x, r3;
          mov r3, register1;
          cmp r5, CMP;
          bne 34;
        end
      end
      else (* nothing *)
  end.
```

And here is the sort of output its compiler must produce (there is no need for the compiler to provide the comments; these are simply to help the reader. Labels can be used as indicated.

```
    MOV   [255], [254]     ; x := r4
    CMP   [255], [254]     ; if (x = r4)
    BNE   L0               ;   then begin
    MOV   [252], [251]     ;     CMP := MOV
    MOV   [255], [250]     ;     x := c
    MOV   R3, [250]        ;     R3 := c
    MOV   [250], 203       ;     c := 203
    BRN   L1               ;   end
L0                         ;   else
    CMP   [255], [253]     ;     if x = r5
    BNE   L2               ;       then
    MOV   [251], [250]     ;         MOV := c
    BRN   L3               ;
L2                         ;       else
    MOV   [251], R3        ;         MOV := r3
L3                         ;
L1                         ;
    MOV   R3, R1           ; r3 := register1
    CMP   [250], [249]     ; if c = d
    BNE   L4               ;   then begin
    MOV   [249], [252]     ;     d := cmp
                           ;     asm
    MOV   R1, R2           ;       mov r1, r2
    MOV   R4, 34           ;       mov r4, 34
    MOV   [34], 34         ;       mov [34], 34
    MOV   [255], [1234]    ;       mov x, base
    MOV   [255], R3        ;       mov x, r3
    MOV   R3, R1           ;       mov r3, register1
    CMP   R5, [252]        ;       cmp r5, CMP
    BNE   34               ;       bne 34
    BRN   L5               ;     end
L4                         ;   else nothing
L5                         ;
```

Naturally, the compiler must be able to recognise semantic errors; here is the sort of listing that should accompany a (deliberately) wrong submission:

```
    1  program example;
    2  (* simple example of wrong mixed language program *)
    3    int
    4      x, r4, r5;        (* some with register names etc *)
    5    bool
    6      CMP, MOV, c2, d;  (* some with operator names *)
    7    int
    8      r3 = r43,         (* attach to register *)
*****           ↑ Error:  No such register
    9      register1 = r1,
   11
   12    begin
   13      x := r4;
   14      if x = r4
   15        then
   16          begin CMP := MOV; x := c; r3 := c; c := 203 end
*****                                 ↑ Error:  Undeclared identifier
   17        else
   18          if x = r5
   19            then MOV := c
*****                       ↑ Error:  Undeclared identifier
   20            else MOV := r3;
   22       base := x;
   23      r4 := base;
   24      if c = d
*****         ↑ Error:  Undeclared identifier
   25        then begin
   26          d := CMP;
   27          asm
   28            mov r1, r2;
   29            mov r4, 34;
   30            mov 34, 34;
   31            mov x;
*****                 ↑ Error:  Too few operands
   32            move x, r3;
*****           ↑ Error:  Invalid opcode
   33            mov r3, register1;
   34            cmp r5, CMP;
   35            bne 34, xx;
*****                 ↑ Error:  Too many operands
*****                     ↑ Error:  Undeclared identifier
   36          end
   37        end
   38        else (* nothing *)
   39    end.
```

Of course, as a programmer trained at the top educational institution in the land you should have learned to use compiler generating tools such as Coco/R, and in spite of incredible time constraints (the deadline for submissions of a prototype compiler is only 24 hours away) you should have little trouble in submitting a working system. So take up the challenge. Being open minded, the firm is prepared to accept implementations written in either Pascal or C++. They are also prepared to accept a fairly simple-minded approach to generating the assembler code - it will suffice to have a system where this is simply directed to the standard output file. And, since they have a licensed copy of Coco/R, they are prepared to accept submissions in the form of a Cocol attribute grammar plus any support modules needed, either on diskette or simply written on paper.

Hint: a complete attribute grammar may turn out to be rather long. For examination purposes it is recommended that you (a) develop a Cocol grammar describing the source language (b) add sufficient of the attributes for an intelligent examiner to be convinced that you understand the form that these would take and how code generation would take place (c) describe at least the interface to support routines that deal with symbol tables (function headings with adequate commentary will suffice). The examners will be looking for evidence of quality and semantic correctness in your solutions, rather than 100% syntactically correct Cocol.

Confine yourself to simple assignment statements and comparisons like those in the example program - that is, where the right hand side of an assignment consists of a single constant or variable, and where comparisons are made for equality only, between operands that are single constants or variables - and consider only `if-then-else` statements where *both* the `then` and `else` clauses are present.

There is also no need to describe the support routines as full classes or units - simply assume that the routines can be coded up within a single `#include` file in the manner familiar from practicals done during the course, or included directly at the head of the grammar itself.