RHODES UNIVERSITY

June Examinations - 1997

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 [120 marks]

A1 Briefly explain the differences between each pair of concepts below:

(a) Self resident compilers and cross compilers
(b) Phrase structure and lexical structure
(c) Scope and extent of identifiers

[9]

A2 Draw a diagram outlining the relationship between the various phases that occur in compilation, and briefly describe the responsibility of each phase.

[10]

A3 Draw T-diagrams that represent the processes that are used in the compiler generator/compiler/interpreter system that you have developed in the practicals of this course.

[8]

A4 A grammar G may be described by a 4-tuple.
(a) Name, and briefly describe, each component of this 4-tuple.
(b) In terms of your notation, define precisely (mathematically) what is meant by a language L defined on the grammar G.
(c) Define precisely (mathematically) what conditions must be satisfied by G to ensure that it is a ″reduced grammar″.

[6, 3, 3]

A5 What do you understand by the concepts ″ambiguous grammars″ and ″equivalent grammars″? Illustrate your answer by giving a simple example of an ambiguous grammar, and of an equivalent non-ambiguous grammar.

[8]

A6 Explain what is meant by the FIRST and FOLLOW sets as applied to grammar analysis, and in terms of these state the rules that a grammar must obey for it to be LL(1).

[8]

A7 You should be familiar with the use of the `CASE` statement in Modula-2. (A few examples of syntactically correct CASE statements appear in appendix A so as to refresh your memory.)

(a) Describe the form of a `CASE` statement in EBNF. You do not need to give productions for *Statement* or for *Expression*, but you should give detailed productions for any other non-terminals you introduce.

(b) Does your set of productions satisfy the LL(1) critera? Explain in detail.

(c) When a Modula-2 compiler translates a `CASE` statement it must check not only that the syntax is correct, but also that various static semantic constraints have been met. Identify at least two of these constraints.

[8, 8, 4]

A8 Appendix B shows an extract from a simple recursive descent compiler that handles compilation of a simple `if - then` statement. Suppose the statement syntax were to be extended to

```
IfStatement = "IF" Condition "THEN" StatementSequence
              [ "ELSIF" Condition "THEN" StatementSequence ]
              [ "ELSE" StatementSequence ]
              "END" .
```

How would the `IfStatement` routine in the compiler have to be changed to accommodate this?

[18]

A9 Appendix C shows a simple recursive Clang program for reading and sorting a list of 6 numbers using a selection sort algorithm.

(a) Indicate how a symbol table might appear when compilation had proceeded as far as the point indicated by the comment.

(b) Discuss the run-time appearance of the stack if this program were to have been translated by the compiler studied in the course, at the stage where the `SelectionSort` routine had made the first recursive call to itself, and had just called the function `MaxPos`. You may assume either of the two possibilities - using a static link, or using a display. (Indicate clearly which you are describing.)

(c) Discuss briefly the advantages and disadvantages of using the static link as opposed to the display method for storage management.

[9, 12, 6]

SECTION B [60 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 EXAMM.ZIP (Modula-2) or EXAMC.ZIP (C++), modify any files that you choose, and then copy all the files back to the blank diskette that will be provided. In this case it would help if you were to highlight your changes with `+++++++++ comments ++++++++ .`

The manufacturers of a range of pocket calculators are anxious to prototype their latest idea. This is for what is essentially a four function calculator with an extensive set of memory cells. Input to the calculator is to be in the form of what programmers would recognise as a list of assignment statements and print statements, terminated by QUIT, for example

```
X := 4
Z := 15.4
Y := 12.5 / (5 * X + 6 * Z)
PRINT X, Y, - Z - X
QUIT
```

However, the calculator has some interesting features:

The user may choose his or her names for memory cells (variables) "on the fly". They do not have to be predeclared - effectively they are "declared" when they are first encountered on the left of an assignment statement. Until such time as they have a value assigned to them, all variables are deemed to have "undefined" values and be of unknown type. If an attempt is made to evaluate an expression containing undefined values, this is to be reported as erroneous. Thus similar input to the above, but of the form

```
X := 4
Y := 12.5 / (5 * X + 6 * Z)
Z := 15.4
PRINT X, Y, - Z - X
```

would be erroneous, as `Z` has not been defined by the time the second assignment is encountered.

The calculator is to be intelligent enough to work in either *integer* or *real* (floating point) mode. Expressions whose operands consist only of *integer* literals (or variables currently known to store *integer* values) and whose operators are limited to `+ - *` and `DIV` are to be evaluated in *integer* mode, while expressions containing any operands of *real* type, and operators including `+ - *` and `/` are to be evaluated in *real* mode. (`DIV` thus denotes familiar *integer* "divide and truncate" operation, `/` denotes floating point division).

When a value is assigned to a variable, the type of that variable is determined. Note that this type may be altered by a later assignment. Thus the sequence

```
ABC := 5 + 12 DIV 3
ABC := - (5.2 + ABC)
```

first assigns the integral value 9 to `ABC`, and then later assigns the value -14.2 to `ABC`, changing the type to *real*.

Although promotion of types in mixed mode expressions such as those given earlier happens automatically much as is done in C++ or Pascal, two functions are provided to allow users explicitly to convert operand values from one type to another. These are exemplified in the expressions below:

```
ANNE := REAL(5 + 6 * 7) - 3.5
            (* ANNE is assigned 47.0 - 3.5 = real 43.5 *)
BOB := INTEGER(6.3 / 2) + 5
            (* BOB is assigned 3 + 5 = integer 8 *)
```

That is, `REAL(i)` simply converts the value of its *integer* argument `i` to the corresponding *real* value, while `INTEGER(r)` converts its *real* argument `r` to the closest *integer* value, truncating where necessary.

It is considered erroneous to apply the `REAL` function to an argument that has been evaluated to be of *real* type, to apply the `INTEGER` function to an argument that is of *integer* type, or to apply the `DIV` operation when either or both operands are of *real* type.

To illustrate the automatic promotion of types, the statements

```
X := 4
Z := 15.4
Y := 12.5 / (150 DIV X + 6 * Z)
```

are thus equivalent to

```
X := 4
Z := 15.4
Y := 12.5 / (REAL(150 DIV X) + REAL(6) * Z)
```

The calculator manufacturers (BPSFH (Pty) Ltd) have heard of Coco/R, and have been led to believe that it can be used to develop a program that will simulate the action of the calculator, reading input expressions from a file, and producing syntactic and semantic analysis of this in a listing file, and the results of the calculations in an output file. They are offering a substantial reward - 33% of a semester credit in Computer Science at a well known university - to anyone who can convince them of this.

Take up the challenge. Being open minded, the firm is prepared to accept implementations written in either Modula-2 or C++, of course. They are also prepared to accept a fairly simple-minded approach to implementing the structure needed to handle the user-defined names for memory locations (for example, limiting this to a linear bounded array). 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.

(The complete attribute grammar may turn out to be rather long. For examination purposes it will suffice to write out enough of the attributes for an intelligent examiner to be convinced that you could actually implement the entire system; be careful in which you choose for illustration, of course.)

Appendix A - Some Modula-2 CASE statements

```
(*   VAR
       integer : INTEGER;
       char : CHAR;
       Color : (Red, Orange, Yellow, Green, Blue, Indigo, Violet); *)

     CASE integer + 5 OF
         1 : Statement1; Statement2;
       | 2 : Statement3;
       | 3 : Statement4; Statement5
     END

     CASE char OF
         'a' : Statement1; Statement2;
       | 'b' : Statement3;
       | 'e' .. 'f', 'x' .. 'z' : Statement4; Statement5
       | CHR(0) .. CHR(31) : Statement6
       ELSE Statement7
     END

     CASE Color OF
       | Green : Statement1; Statement2;
       |
       | Red   : Statement3;
       |
       | Yellow .. Violet : Statement4; Statement5
       ELSE Statement6
     END
```

Appendix B An IfStatement parser

In C++

```
void PARSER::IfStatement(symset followers)
// IfStatement = "IF" Condition "THEN" StatementSequence "END" .
{ CGEN_labels testlabel;
  GetSym();
  Condition(symset(SCAN_thensym, SCAN_dosym) + followers);
  CGen->jumponfalse(testlabel, CGen->undefined);
  if (SYM.sym == SCAN_thensym)
    GetSym();
  else
    { Report->error(23); if (SYM.sym == SCAN_dosym) GetSym(); }
  StatementSequence(followers);
  CGen->backpatch(testlabel);
  Accept(SCAN_endsym, 24);
}
```

In Modula-2

```
PROCEDURE IfStatement;
(* IfStatement = "IF" Condition "THEN" StatementSequence "END" . *)
  VAR
    TestLabel : CGEN.LABELS;
  BEGIN
    SCAN.GetSYM; Condition(SYMSET{ThenSym, DoSym} + Followers);
    CGEN.JumpOnFalse(TestLabel, CGEN.undefined);
    IF SYM.Sym = ThenSym
      THEN SCAN.GetSYM
      ELSE Error(23); IF SYM.Sym = DoSym THEN SCAN.GetSYM END
    END;
    StatementSequence(Followers);
    CGEN.BackPatch(TestLabel);
    Accept(EndSym, 24);
  END IfStatement;
```

Appendix C A sorting program in Clang

```
PROGRAM SortDemo;

  PROCEDURE SelectionSort (List [], N);
  (* Sorts N elements of List[0..N] into order *)

    VAR
      T, M;

    FUNCTION MaxPos (List[], N);
    (* Returns position of maximum element in List[0..N] *)
      VAR
        I, Max;
      BEGIN
        (* +++++ Discuss run time storage management when this point is reached for the second time *)
        I := 1; Max := 0;
        WHILE I <= N DO BEGIN
          IF List[I] > List[Max] THEN Max := I;
          I := I + 1
        END;
        RETURN Max
        (* +++++ Discuss symbol table structure when parsing reaches this point *)
      END;

    BEGIN
      IF N > 0 THEN BEGIN
        M := MaxPos(List, N);
        T := List[N]; List[N] := List[M]; List[M] := T;
        SelectionSort(List, N-1)
      END
    END;

    VAR
      Data[5], I;
    BEGIN
      I := 0; WHILE I <= 5 DO BEGIN Read(Data[I]); I := I + 1 END;
      SelectionSort(Data, 5);
      I := 0; WHILE I <= 5 DO BEGIN Write(Data[I]); I := I + 1 END;
    END.
```