RHODES UNIVERSITY

November Examinations - 1999

Computer Science 301 - Paper 1

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

Time 3 hours
Marks 180
Pages 8 (please check!)

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

(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included the full text of Section B. During the examination, candidates were given machine executable versions of the Coco/R compiler generator, and access to a computer and machine readable copies of the questions.)

Section A [110 marks]

1 (a) Explain what is meant by the FIRST and FOLLOW sets as applied to grammar analysis, and give a concise (mathematical) definition of set membership for each of these. [4]

(b) State the rules that a grammar must obey for it to be LL(1). Do not write vague English; describe the rules in concise mathematics. [6]

2 EBNF and BNF are notations for defining productions in context free grammars. In EBNF, as you should know, "repetition" and "optionality" may be expressed using metabracket notation, whereas in BNF these features are expressed using recursion and selection. The Cocol grammar below effectively uses EBNF notation to describe the form EBNF productions can take.

(a) What do the acronyms "EBNF" and "BNF" stand for? [2]

(b) How would you change the `PRODUCTIONS` section of the Cocol specification below so that it used EBNF conventions, but described *only* the form of BNF productions? [4]

(c) How would you change the `PRODUCTIONS` section of the Cocol specification below so that it *only* used BNF conventions, but described the form of EBNF productions? [6]

(d) Do the productions that you have written for (c) result in an LL(1) compliant grammar? Give reasons for your answer. [4]

```
COMPILER EBNF
/* Describe simple EBNF productions, using EBNF conventions */

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  lowline  = "_".
  digit    = "0123456789".
  noquote1 = ANY - "'".
  noquote2 = ANY - '"'.

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

TOKENS
  nonterminal = letter {letter | lowline | digit} .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .
PRODUCTIONS
   EBNF       = { Production } EOF .
   Production = nonterminal "=" Expression "." .
   Expression = Term { "|" Term } .
   Term       = [ Factor { Factor } ] .
   Factor     =   nonterminal | terminal | "(" Expression ")"
                | "[" Expression "]" | "{" Expression "}" .
END EBNF.
```

3 Palindromes are strings that read the same from either end, such as "abba", "aha", "noon" or "wow". Presumably one can write grammars to describe palindromes. Here are a few attempts to write productions

to do this for palindromes that consist only of the letters "a" and "b".

```
Grammar 1:
    A = "a" A "a" | "b" A "b" .

Grammar 2:
   A = "a" A "a" | "b" A "b"  | "a" | "b" .

Grammar 3:
   A = B A B | ε .
   B = "a" | "b" .
```

Some (or maybe all) of these grammars do not describe the language of such palindromes adequately. Identify which do not, giving reasons for your answer, and present a set of productions that you believe is correct. Is it possible to find an LL(1) compliant grammar for this language. If not, why not? [12]

4 Here is a true story: A few weeks ago, one of the many Coco/R users in the world emailed me with a suggestion. He was unimpressed with the way in which one had to write specifications like

```
CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit = "0123456789".
```

and suggested that we add a feature to allow (as an alternative)

```
CHARACTERS
  letter = CHARS("A" .. "Z") + CHARS("a" .. "z") .
  digit = CHARS("0" .. "9") .
```

Since I have access to the sources of Coco, which is a self-compiling system, I should be able to bootstrap the new system quite easily, (by adding to one of the productions in the file CR.ATG).

(a) What do you understand by the term *self-compiling compiler*? [2]

(b) Draw T-diagrams showing what I should do to produce the new Coco/R executable for distribution on our web page. [8]

5. *Pragmas* and *comments* often appear in source code submitted to compilers. What property do pragmas and comments have in common, and what is the semantic difference between them? [5]

6. What do you understand by the term *short circuit semantics* as applied to the evaluation of Boolean expressions? Illustrate your answer by means of two specimens of code that might correspond to a simple Boolean expression of your choice. [5]

7. Consider the grammar in the appendix for a (familiar) tiny C-like subset language, and the example of a program in that language.

(a) The grammar for ExtaC is "context free". What do you understand by the term "context free" as applied to compiler theory? (Give a concise, mathematical definition, not a mini-essay!) [3]

(b) Show that you understand what is meant by the concept of "scope" by indicating clearly which identifiers (give their names and their types) you understand to be "in scope" at the point marked by the comment `/* what is in scope at this point? */` in the specimen program below. [4]

(c) Detach the diagram provided on page 8, and by completing it and writing appropriate notes, discuss how the run-time memory management could be handled for programs of this sort. In the course of your discussion elaborate on the ideas of *stack frames, dynamic links, static links*, and *stack* and *base pointers*, and illustrate what the layout of memory might be when the program below reaches the point indicated for the second time (the program is recursive, and would, of course, never terminate, but ignore that for the purposes of this discussion). [20]

```
/* Silly example of ExtaC - not claimed to have any real dynamic semantics */

  char x, y, z;
  bool a, b;

  void One (void);  /* prototype */

  void Two (void) {
    const int z = -4;
    bool y;
    int x;
                  /* what is in scope at this point? */
    One();
    x = 40 / 5;
  }

  int p, q, r;
  const bool EasyExam = true || false;

  void One (void) {
                  /* discuss run-time memory allocation when this point is
                     reached for the second time */
    Two();
  }

  void main (void) {
    char p, q, r;
    p = q;
    One();
  }
```

8. Brinch Hansen introduced an extended form of the `WHILE` loop into his experimental language Edison:

```
WhileStatement  =  "while" Expression "do" Statement
                   { "else" Expression "do" Statement } .
```

The dynamic semantics of this peculiar construct are that `Expressions` are evaluated one at a time in the order written until one is found to be true, when the corresponding `Statement` is executed, after which the process of evaluating conditions is repeated. If no `Expression` is true, the loop terminates.

Assuming that you want to incorporate this into a language like ExtaC, and to target a stack machine similar to the one you have met on the course, how would you attribute the production to handle code generation? [10]

9. In the Clang compiler with which you are familiar, you will recall that the language allowed for constant declarations:

```
ConstDeclarations = "CONST" OneConst { OneConst } .
OneConst          = Ident "=" ( Number | "TRUE" | "FALSE" ) ";" .
```

(The attributed grammar that processed these is presented in an appendix to refresh your memory).

Suppose we wished to extend the language so that `CONST` could also be used to introduce synonyms (alternative names) for previously-declared variables and constants, for example

```
CONST
  Max = 10;
VAR
  X, Y[10];
CONST                (* this time to act as a renaming facility *)
  MyMax = Max;       (* MyMax and Max are really the same constant *)
  YourX = X;         (* YourX is another name for X *)
  List  = Y;         (* List is another name for Y *)
BEGIN
  X := Max + MyMax;  (* X would be assigned the value 10 + 10 *)
  List[3] := Y[4];   (* replace the 3rd element of the array by the 4th element *)
```

Alter the specification and attributes of the `OneConst` production to achieve this. [10]

Section B [75 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.

We have a minor crisis in the Department. Because the experienced staff member who normally gives the Boolean Algebra section of our course has just been taken ill, we have had to ask a new lecturer to take over the course at very short notice, and he needs help …

We assume that you are familiar with the basic concepts of Boolean algebra, which is defined on a set of operands, say $B = \{a, b, \ldots\}$, two binary infix operators . and + (AND and OR), and a unary postfix operator ' (NOT) .

As you will recall, these operators have a precedence ordering. NOT takes precedence over AND, which takes precedence over OR. This means that an expression written without parentheses, such as $a.b + c$ (or, equivalently, a AND b OR c) means the same as the parenthesized expression $(a.b) + c$, and does **not** mean the same as the parenthesized expression $a.(b + c)$. Sometimes the AND operation becomes implicit - as in simple algebra where *"a b"* can be read as *"a* times *b"*, so in Boolean algebra *"a b"* can be read as *"a* AND *b"*

Other symbols are sometimes used for the operators. (Expressed as an apostrophe ′ , the NOT operator is one of the very few mathematical operators that is written *after* its operand, as what is called a *postfix operator*. Another example of a postfix operator is the "factorial" one, as in *n!*). In computer language terms it is usual to find prefix operators preferred to postfix ones, and to avoid the use of . and +, which tend to get confused with decimal points and addition signs. Hence one finds the use of the full words (as in Pascal and Modula-2) or the use of other symbols entirely, such as & to mean AND, | to mean OR, and ! or ~ to mean NOT.

Here is the same Boolean expression written in several different ways, using these various notations, with parentheses used in the expressions on the right to make the precedence quite explicit:

A AND B OR NOT C AND D	(A AND B) OR ((NOT C) AND D)
$a.b + c'.d$	$(a.b) + (c'.d)$
$a\ b + c'\ d$	$(a\ b) + (c'\ d)$
$a \,\&\, b \mid \tilde{}c \,\&\, d$	$(a \,\&\, b) \mid ((\tilde{}c) \,\&\, d)$
$a \,\&\, b \mid !c \,\&\, d$	$(a \,\&\, b) \mid ((!c) \,\&\, d)$

When beginners are taught Boolean algebra and combinational circuits in introductory Computer Science courses, they are often introduced to the concept of a truth table, like those given below (sometimes these tables are expressed in terms of 0′s and 1′s; sometimes in terms of `TRUE` and `FALSE`).

X	NOT X
FALSE	TRUE
TRUE	FALSE

ONE	TWO	ONE XOR TWO
0	0	0
0	1	1
1	0	1
1	1	0

Such tables are easy enough to derive by hand, though it is tedious to do so for expressions that involve more than two inputs, and the process is error prone. On Monday the new lecturer is scheduled to teach students about truth tables, and he is desperately looking for a tool that will help him produce these in a hurry for arbitrarily complex expressions. He wants to be able to feed in a list of (independent) functions, for example

```
BOOLEAN F(x) = x';
BOOLEAN f(x, y) = x | y;
xor(one, two) = one & two' or one' & two;
CanReturn(S1, S2, S3, Excluded) = (S1 S2 OR S2 S3 + S3 S1) AND ò Excluded;
BOOLEAN Equality(x) = x ' ' & ! (NOT x);
NUMERIC Silly(Twit) = 1 . 0' + Twit;
BOOLEAN Silly(Twit) = TRUE AND NOT FALSE OR Twit
```

each one defining some function of its explicit arguments, and separated one from the next by the traditional semicolons. For each function a table should be produced showing the value of the function for each combination of the possible values of its parameters. If present, the keyword `BOOLEAN` indicates that the table should be produced in terms of combinations of `TRUE` and `FALSE`; the absence of a keyword, or the explicit keyword `NUMERIC` indicates that the table should be produced in terms of combinations of 0′s and 1′s. Notice that the expressions defining the functions:

(a) may only refer to variables in the parentheses of the function header

(b) may refer to these variables more than once in an expression, as illustrated by `CanReturn`

(c) may not refer to other functions and

(d) must be able to accept any of the equivalent tokens for operators within an expression (as exemplified in `CanComplete`, where one sees word tokens `AND` and `OR`, the alternative `~` used for `NOT`, and the alternative + used for `OR`).

Of course, as a senior undergraduate well versed in the use of compiler generating tools such as Coco/R, and in spite of these incredible time constraints, you should have little trouble in producing a system for him to use by Saturday evening, well in advance of Monday morning. After all, you have learned to generate code for simple stack machines, and still have that copy of a Clang compiler to turn to for useful ideas! So take up the challenge.

Being human like the rest of us, the lecturer wants a tool that will check his expressions for syntactic and semantic correctness. He is prepared to accept a fairly simple-minded approach to generating the results - it will suffice to have a system where these are simply directed to the standard output file. And, since he has a licensed copy of both the C++ and Pascal versions of Coco/R, he is prepared to accept submissions in the form of a Cocol attribute grammar using either language, plus any support modules needed, either on diskette or simply written on paper.

Appendix: Extracts from the Clang compiler specification relating to the declaration of constants.

C++ version:

Grammar:

```
ConstDeclarations =  "CONST" OneConst { OneConst } .
OneConst
=                              (. TABLE_entries entry; .)
   Ident<entry.name>           (. entry.idclass = TABLE_consts; .)
   WEAK "="
   ( Number<entry.c.value>     (. entry.type = TABLE_ints; .)
     | "TRUE"                  (. entry.type = TABLE_bools; entry.c.value = 1; .)
     | "FALSE"                 (. entry.type = TABLE_bools; entry.c.value = 0; .)
   ) ";"                       (. Table->enter(entry); .) .
```

Definitions of entries in the symbol table handler:

```
const int TABLE_alfalength = 15; // maximum length of identifiers
typedef char TABLE_alfa[TABLE_alfalength + 1];

enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs };
enum TABLE_types {TABLE_none, TABLE_ints, TABLE_chars, TABLE_bools };

struct TABLE_entries {
  TABLE_alfa name;             // identifier
  TABLE_idclasses idclass;     // class
  TABLE_types type;
  union {
    struct {
      int value;
    } c;                       // constants
    struct {
      int size, offset;
      bool scalar;
    } v;                       // variables
  };
};
```

Pascal version

Grammar:

```
ConstDeclarations =  "CONST" OneConst { OneConst } .
OneConst
                               (. VAR Entry : TABLE.ENTRIES; .)
=  Ident<Entry.Name>           (. Entry.IDClass := Consts  .)
   WEAK "="
   ( Number<Entry.Value>       (. Entry.Type := ints; .)
     | "TRUE"                  (. Entry.Type := bools; Entry.Value := 1; .)
     | "FALSE"                 (. Entry.Type := bools; Entry.Value := 0; .)
   ) ";"                       (. TABLE.Enter(Entry) .) .
```

Definitions of entries in the symbol table handler:

```
TYPE
  ALFA      = STRING[AlfaLength];
  IDCLASSES = (Consts, Vars, Progs);
  TYPES     = (none, ints, chars, bools);
  ENTRIES   =
    RECORD
      Name : ALFA;
      Tipe : TYPES;
      CASE IDClass : IDCLASSES OF
         Consts :
          ( Value : INTEGER );
         Vars :
          ( Size, Offset : INTEGER;
            Scalar : BOOLEAN )
    END;
```

Appendix B : Context free grammar for ExtaC

```
COMPILER ExtaC

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

CHARACTERS
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .

PRODUCTIONS
  ExtaC        = { Library } { Declaration } .
  Library      = "#include" "<" identifier ">" .
  Declaration  = Variables | Constant | Function .
  Variables    = Type identifier { "," identifier } ";" .
  Type         = "int" | "bool" | "char" .
  Constant     = "const" [ Type ] identifier "=" Expression ";" .
  Function     = "void" identifier "(" FormalParams ")"
                 (   ";"                      /* prototype */
                   | "{" { Statement } "}"     /* definition */
                 ) .
  FormalParams = [ Type identifier { "," Type identifier } | "void" ] .
  Statement    = Declaration | AssignOrCall .
  AssignOrCall = identifier ( "=" Expression | "(" ActualParams ")" ) ";" .
  ActualParams = [ Expression { "," Expression } ] .
  Expression   = Term { AddOp Term } .
  Term         = Factor { MulOp Factor } .
  Factor       = [ "-" | "!" ] ( identifier | number | "(" Expression ")" ) .
  Prototype    = "void" identifier "(" FormalParams ")" ";" .
  AddOp        = "+" | "-" | "||" .
  MulOp        = "*" | "/" | "&&" .
END ExtaC.
```