RHODES UNIVERSITY

November Examinations - 2000

Computer Science 301 - Paper 1

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

Time 3 hours
Marks 180
Pages 7 (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, access to a computer and machine readable copies of the questions.)

Section A [95 marks]

1. Formally, a grammar G is 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:

   $$\alpha \rightarrow \beta \text{ where } \alpha \in (N \cup T)^* N (N \cup T)^* \text{ , } \beta \in (N \cup T)^*$$

   and formally we can define a language $L(G)$ produced by a grammar G by the relation

   $$L(G) = \{ \sigma \mid \sigma \in T^* ; S \Rightarrow^* \sigma \}$$

   In terms of this notation, express **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by [3 marks each]

   (a) A context-sensitive grammar

   (b) A context-free grammar

   (c) Reduced grammar

   (d) FIRST(A) where $A \in N$

   (e) FOLLOW(A) where $A \in N$

   (f) Nullable productions

2. 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 marks]

3. Here is a Cocol description of simple mathematical expressions:

```
COMPILER Expression  $XNC  /* expr.atg */
CHARACTERS
  digit      = "0123456789" .
TOKENS
  Number     = digit { digit } .
PRODUCTIONS
  Expression = Term { "+" Term  | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     = Number .
END Expression.
```

Two CS3 students were arguing very late at night / early in the morning about a prac question which read "extend this grammar to allow you to have leading unary minus signs in expressions, as exemplified by `-5 * (-4 + 6)`, and make sure you get the precedence of all the operators correct". One student suggested that the productions should be changed to read (call this GRAMMAR A)

```
Expression = [ "-" ] Term { "+" Term  | "-" Term } .   (* change here *)
Term       = Factor { "*" Factor | "/" Factor } .
Factor     = Number .
```

while the other suggested that the correct answer would be (call this GRAMMAR B)

```
Expression = Term { "+" Term  | "-" Term } .
Term       = Factor { "*" Factor | "/" Factor } .
Factor     = Number | "-" Factor .                    (* change here *)
```

By drawing the parse trees for the string `- 12 * 4 + 5` try to decide which student was correct. Or were they both correct? If you conclude that the grammars can both accept that particular expression string, are the grammars really equivalent, or can you give an example of an expression that GRAMMAR A would accept but GRAMMAR B would reject (or vice-versa)? [12 marks]

4. The well-studied grammar for a preliminary version of Topsy is provided in an appendix to this paper. (`TOPSY0.ATG`) Suppose we wanted to add C++ like "switch statements" to the language.

   (a) What form would the modifications to the syntax have to take? (Write out only the altered productions). Here are some examples of switch statements to help you decide (`Q4.TXT`): [6 marks]

```
switch (a + b) {
  case 3 :
  case 4 : a = b; c = d;
  case 5 : while (b >= c) b--; break;
  default : x = y;
}

switch (f) ; // a very trivial one!

switch (a < b) {
  case true : x = y;
  case false : y = x;
}

switch (x)  // yes, this is quite legal!
  default :
    if (prime(x))
      case 2: case 3: case 5: case 7: processPrime(x);
    else
      case 4: case 6: case 8: case 9: case 10: processComposite(x);
```

   (b) Switch statements are non-trivial. There are a few constraints on them that you may find easier to express as semantic constraints, rather than fight with ever more convoluted syntax to try in vain to achieve the same effect. Suggest what at least three of these constraints might be (you do not have to write an attributed grammar to impose the constraints; simply describe them in English). [6 marks]

5. Here is a set of productions that describes the weekly misery of a member of this class engaged on a practical assignment:

```
PRAC     = Handouts { Task } [ Essay ] "Submit" .
Handouts = "PracSheet" { "HintSheet" } .
Task     = "Attempt" { Help  "Attempt" }  .
Essay    = "CorelFormat" | "MSWordFormat" | "ASCIIFormat" .
Help     = [ "Pat" | "Holger" | "Barry" | "Colin" | "Guy" ] [ "Pat" ] .
```

   Analyse this set of productions and discuss whether they conform to the LL(1) restrictions on grammars. If they do not, explain carefully where and why this happens. [12 marks]

6. Suppose that you are writing a recursive descent parser for a language *L* described by a grammar *G* known to conform to the LL(1) restrictions. As has often been claimed, writing such a parser is very easy if one can assume that the users of the language *L* will never make coding errors, but this is wishful thinking! Write a brief essay on how one might modify the parsing routine for a non-terminal *A* of the grammar *G* so as to incorporate simple but effective error recovery. [15 marks]

7. The Clang and Topsy languages used in this course are, of course, remarkably similar. One year I developed a lot of simple Clang programs to test a Clang compiler, and then the next year found I needed the equivalent Topsy programs to test a Topsy compiler. Being a lazy soul I realised that all I needed to do was to use Coco/R to develop a sort of pretty printer that would convert my existing Clang programs by changing keywords into lower case, inserting parentheses around conditions, and so on. At the outset I sketched a few T-diagrams to describe this process. Draw such a set of T diagrams. (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.) [10 marks]

8. Here is a true story. A few weeks 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 very astutely that this was very tedious, and also 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 order but subject to the restriction that each should appear once.

   Write the appropriate parts of a Cocol specification that will describe the situation and check that the restrictions are correctly met. (Restrict your answer to the case of 4 derived non-terminals, as above.) [8 marks]

Section B [85 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.

Most computer languages provide simple, familiar, notations for handling arithmetic, character and Boolean types of data. Variables, structures and arrays can be declared of these basic types; they may be passed from one routine to another as parameters, and so on.

Some languages, notably Pascal, Modula-2, C, C++, and Ada, allow programmers the flexibility to define what are often known as *enumeration types*, or simply *enumerations*. Here are some examples to remind you of this idea:

```
TYPE  (* Pascal or Modula-2 *)
  COLOURS = ( Red, Orange, Yellow, Green, Blue, Indigo, Violet );
  INSTRUMENTS = ( Drum, Bass, Guitar, Trumpet, Trombone, Saxophone, Bagpipe );
VAR
  Walls, Ceiling, Roof : COLOURS;
  JazzBand : ARRAY [0 .. 40] OF INSTRUMENTS;
```

or the equivalent

```
typedef  /* C or C++ */
  enum { Red, Orange, Yellow, Green, Blue, Indigo, Violet } COLOURS;
typedef
  enum { Drum, Bass, Guitar, Trumpet, Trombone, Saxophone, Bagpipe } INSTRUMENTS;
COLOURS Walls, Ceiling, Roof;
INSTRUMENTS  JazzBand[41];
```

Sometimes the variables are declared directly in terms of the enumerations:

```
VAR  (* Pascal or Modula-2 *)
  CarHireFleet : ARRAY [1 .. 100] OF ( Golf, Tazz, Sierra, BMW316 );

enum CARS { Golf, Tazz, Sierra, BMW316 } CarHireFleet[101];  /* C or C++ */
```

The big idea here is to introduce a distinct, usually rather small, set of values which a variable can legitimately be assigned. Internally these values are represented by small integers - in the case of the `CarHireFleet` example the "value" of `Golf` would be 0, the value of `Tazz` would be 1, the value of `Sierra` would be 2, and so on.

In the C/C++ development of this idea the enumeration, in fact, results in nothing more than the creation of an implicit list of `const int` declarations. Thus the code

```
enum CARS { Golf, Tazz, Sierra, BMW316 } CarHireFleet[101];
```

is semantically completely equivalent to

```
const int Golf = 0; const int Tazz = 1;
const int Sierra = 2, const int BMW316 = 3;
int CarHireFleet[101];
```

and to all intents and purposes this gains very little, other than possible readability; an assignment like

```
CarHireFleet[N] = Tazz;
```

might mean more to a reader than the semantically identical

```
CarHireFleet[N] = 1;
```

In the much more rigorous Pascal and Modula-2 approach one would not be allowed this freedom; one would be forced to write

```
CarHireFleet[N] := Tazz;
```

Furthermore, whereas in C/C++ one could write code with rather dubious meaning like

```
CarHireFleet[4] = 45;             /* Even though 45 does not correspond to any known car! */
CarHireFleet[1] = Tazz / Sierra;  /* Oh come, come! */
Walls = Sierra;                   /* Whatever turns you on is allowed in C++ */
```

in Pascal and Modula-2 one cannot perform arithmetic on variables of these types directly, or assign values of one type to variables of an explicitly different type, or assign values that are completely out of range. In short, the idea is to promote "safe" programming - if variables can meaningfully only assume one of a small set of values, the compiler (or run-time system) should prevent the programmer from writing (or executing) meaningless statements.

Clearly there are some operations that could have sensible meaning. Looping and comparison statements like

```
if (Walls == Indigo) Redecorate(Blue);
```

or

```
for (Roof = Red; Roof <= Violet; Roof++) DiscussWithNeighbours(Roof);
```

or

```
if (JazzBand[N] >= Saxophone) Shoot(JazzBand[N]);
```

might reasonably be thought to make perfect sense - and would be easy to "implement" in terms of the underlying integer values.

In fact, the idea of a limited enumeration is already embodied in the standard character and Boolean types - type Boolean is really the enumeration of the values `{0, 1}` identified as `{false, true}`, although this type is so common that the programmer is not required to declare the type explicitly. Similarly, the character type is really an enumeration of a sequence of (typically) ASCII codes, and so on.

Although Pascal and Modula-2 forbid programmers from abusing variables and constants of any enumeration types that they might declare, the idea of "casting" allows them to bypass the security where necessary. The standard function `ORD(x)` can be applied to a value of an enumeration type to do nothing more than cheat the compiler into extracting the underlying integral value. This, and the inverse operation of cheating the compiler into thinking that it is dealing with a user-defined value when you want to map it from an integer are exemplified by code like

```
IF (ORD(Bagpipe) > 4) THEN .....
Roof := COLOURS(I + 5);
```

Rather annoyingly, in Pascal and Modula-2 one cannot `READ` and `WRITE` values of enumeration types directly - one has to use these casting functions to achieve the desired effects.

Enumerations are a "luxury" - clearly they are not really *needed*, as all they provide is a slightly safer way of programming with small integers. Not surprisingly, therefore, they are not found in languages like Java (simplified from C++) or Oberon (simplified from Modula-2).

During this course you have studied and extended a compiler for a small language, Topsy++, which has syntax similar to C++, but in the implementation of which we have repeatedly stressed the ideas and merits of safe programming. In the examination "kit" you will find the tools we have used to develop this compiler, namely a C++ or Pascal version of the Coco/R compiler generator, frame files, the attributed grammar for Topsy++, and the support modules for the symbol table handler, code generator and interpreter for the version of Topsy++ as it was extended during the final stages of the laboratory work.

How would you add the ability to define enumeration types in Topsy++ programs and to implement these types, at the same time providing safeguards to ensure that they could not be abused? It will suffice to restrict your answer to use a syntax where the variables are declared directly in terms of the enumerations, that is, do not try to handle the `typedef` (or `TYPE`) form of declaration.

A completely detailed solution to this reasonably large exercise might take the form of a complete set of attribute grammar and supporting module source files, and it is highly likely that you cannot provide these in full in the time available in the examination session.

The examiners will be looking for evidence that you

- are familiar with the use of Cocol and can write appropriate productions and add attributes;
- appreciate the use of a symbol table for maintaining contextual information;
- are able to implement type and range checking;
- are able to provide appropriate compile time and run time error detection.

During the formal examination period you would be advised to concentrate simply on describing the changes and alterations to the attribute grammar and support files in as much detail as time permits, preferably by providing extracts of appropriate sections of code. Listings of the attribute grammar for Topsy++ will be available to

candidates who require them. However, in the 24 hours available before the formal examination period, by careful study of the example Topsy++ programs in the exam kit, and taking advantage of the opportunity to discuss your approach with other members of the class, you should be able to get a long way towards making this little language "absolutely perfect". (As I said early one Friday morning - any language would be absolutely perfect if it had just one more feature!)

You may wish to read up a little more on enumeration types as they are used in languages like Modula-2. An essay on these can be found on the course WWW page by following a fairly obvious link.

A typical test program in the kit reads:

```
   void main (void) {
   // Illustrate some simple enumeration types in extended Topsy++
   // Some valid declarations
     enum DAYS { Mon, Tues, Wed, Thurs, Fri, Sat, Sun } Today, Yesterday;
     enum WORKERS { BlueCollar, WhiteCollar, Manager, Boss } Staff[12];
     int i, j, k, PayPacket[12];
     const pay = 100;
     bool rich;
   // Some invalid declarations - your system should be able to detect these
     enum DEGREE { BSc, BA, BCom, MSc, PhD };              // No variables declared
     enum FRUIT { Orange, Pear, Banana, Grape } Favourite; // This is okay
     enum COLOURS { Red, Orange, Green } Paint;            // Orange not unique
   // Some potentially sensible statements
     Today = Tues;
     Yesterday = Mon;                                      // That follows!
     if (Today < Yesterday) cout << "Compiler error";      // Should not occur
     Today++;                                              // Working past midnight?
     if (Today != Wed) cout << "another compiler error";
     int totalPay = 0;
     for (Today = Mon; Today <= Fri; Today++) totalPay = totalPay + pay;
     for (Today = Sat; Today <= Sun; Today++) totalPay = totalPay + 2 * pay;
     rich = Staff[i] > Manager;
     Yesterday = DAYS(int(Today) - 1);  // unless Today is Mon
   // Some meaningless statements - your system should be able to detect these
     Sun++;                   // Cannot increment a constant
     Today = Sun; Today++;    // There is no day past Sun
     if (Today == 4)          // Invalid comparison - type incompatibility
       Boss = rich;           // Invalid assignment - type incompatibility
     Manager = Boss;          // Cannot assign to a constant
     PayPacket[Boss] = 1000; // Incompatible subscript type
   }
```

Appendix - Topsy grammar - unattributed

```
COMPILER Topsy  $XNC  /* Topsy0.atg */
/* Topsy level 1 grammar - no procedures, functions, parameters
   (This does not include the extensions in Topsy++).
   Grammar only - no code generation or constraint analysis
   P.D. Terry, Rhodes University, 1996 */

PRAGMAS
  DebugOn    = "$D+" .

CHARACTERS
  cr         = CHR(13) .
  lf         = CHR(10) .
  back       = CHR(92) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  graphic    = ANY - '"' - "'" - "\" - cr - lf - CHR(0) .

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

TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  string     = '"' { graphic | back graphic | "\'" | back back | '\"' } '"' .
  character  = "'" ( graphic | back graphic | "\'" | back back | '\"' ) "'" .

PRODUCTIONS
  Topsy = "void" "main" "(" "void" ")" Block .

  Block = "{" { Declaration | Statement } "}" .
```

```
  Declaration = ConstDeclaration | VarDeclarations .

  Statement
  =    Block           | Assignment
     | IfStatement     | WhileStatement
     | ForStatement    | RepeatStatement
     | ReadStatement   | WriteStatement
     | ReturnStatement | EmptyStatement .

  ConstDeclaration
  =  "const" identifier "=" ( number | character | "true" | "false" ) ";" .

  VarDeclarations = ( "int" | "bool" | "char" ) OneVar { "," OneVar } ";" .

  OneVar = identifier [ ArraySize | "=" Expression ] .

  ArraySize = "[" number "]" .

  Assignment = Variable ( "=" Expression | "++" | "--" ) ";" .

  Variable = Designator .

  Designator = identifier [ "[" Expression "]" ] .

  IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] .

  WhileStatement = "while" "(" Condition ")" Statement .

  RepeatStatement = "do" { Statement } "until" "(" Condition ")" ";" .

  Condition = Expression .

  ForStatement
  =  "for" "(" identifier "=" Expression ";" Condition [ ";" Epilogue ] ")"
     Statement .

  Epilogue = identifier ( "=" Expression | "++" | "--" ) .

  ReadStatement = "cin" ">>" Variable { ">>" Variable } ";" .

  WriteStatement = "cout" "<<" WriteElement { "<<" WriteElement } ";" .

  WriteElement = string | Expression .

  ReturnStatement = "return" ";" .

  EmptyStatement = ";" .

  Expression = SimpleExpr [ RelOp SimpleExpr ] .

  SimpleExpr = ( "+" Term | "-" Term | Term ) { AddOp Term } .

  Term = Factor { MulOp Factor } .

  Factor =   Designator | number | character | "true" | "false"
           | "!" Factor | "(" Expression ")" .

  AddOp = "+" | "-" | "||" .

  MulOp = "*" | "/" | "%" | "&&" .

  RelOp = "==" | "!=" | "<" | "<=" | ">" | ">=" .

END Topsy.
```