RHODES UNIVERSITY

June Examinations - 2002 (Solutions)

Computer Science 301 - Paper 1

Examiners:
Prof P.D. Terry
Prof E.H. Blake

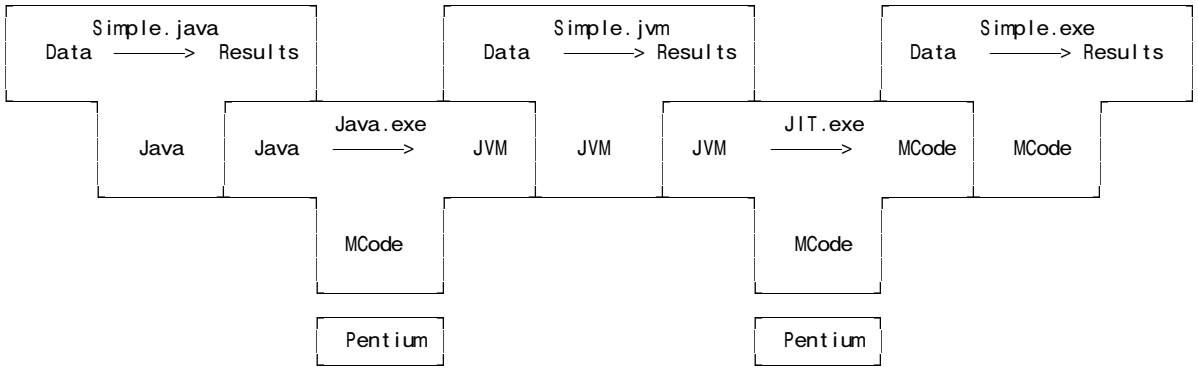
Time 3 hours
Marks 180
Pages 10 (please check!)

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. A Java compiler might typically achieve its goals by compiling first to so-called Java byte code (an interpretable pseudo-assembly code) and then using a complementary JIT ("just in time") assembler to finish the job. Complete the T diagram representation of how such an arrangement would handle the compilation of a simple Java program. [8 marks]



2. 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:

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

   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

The number of symbols in the string on the left of any production is less than or equal to the number of symbols on the right side of that production. In fact, to qualify for being of type 1 rather than of a yet more restricted type, it is necessary for the grammar to contain at least one production with a left side longer than one symbol.

Productions in type 1 grammars (context-sensitive) are of the general form

$$\alpha \rightarrow \beta \qquad \text{with } |\alpha| \leq |\beta|,\ \alpha \in (N \cup T)^* N (N \cup T)^*,\ \beta \in (N \cup T)^+$$

In another definition, productions are required to be limited to the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta \qquad \text{with } \alpha, \beta \in (N \cup T)^*,\ A \in N^+,\ \gamma \in (N \cup T)^+$$

although examples are often given where productions are of a more general form, namely

$$\alpha A \beta \rightarrow \zeta \gamma \xi \qquad \text{with } \alpha, \beta, \zeta, \xi \in (N \cup T)^*,\ A \in N^+,\ \gamma \in (N \cup T)^+$$

(b) A context-free grammar

A grammar is context-free if the left side of every production consists of a single non-terminal, and the right side consists of a non-empty sequence of terminals and non-terminals, so that productions have the form

$$\alpha \rightarrow \beta \qquad \text{with } |\alpha| \leq |\beta|,\ \alpha \in N,\ \beta \in (N \cup T)^+$$

that is

$$A \rightarrow \beta \qquad \text{with } A \in N,\ \beta \in (N \cup T)^+$$

(c) Reduced grammar

A context-free grammar is said to be reduced if, for each non-terminal B we can write

$$S \Rightarrow^* \alpha B \beta$$

for some strings α and β, and where

$$B \Rightarrow^* \gamma$$

for some $\gamma \in T^*$.

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

The FIRST set is best defined by

$$a \in \text{FIRST}(A) \quad \text{if } A \Rightarrow^+ a\zeta \quad (A \in N;\ a \in T;\ \zeta \in (N \cup T)^*)$$

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

It is convenient to define the **terminal successors** of a non-terminal A as the set of all terminals that can follow A in any sentential form, that is

$$a \in \text{FOLLOW}(A) \quad \text{if } S \Rightarrow^* \xi A a \zeta \quad (A, S \in N;\ a \in T;\ \xi, \zeta \in (N \cup T)^*)$$

(f) Nullable productions

If for some string σ it is possible that

$$\sigma \Rightarrow^* \varepsilon$$

then we say that σ is *nullable*. A non-terminal L is said to be nullable if it has a production whose *definition* (right side) is nullable.

3. Long ago, the Romans represented 1 through 10 as 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) Why is this particular grammar ambiguous? [2 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. The skeleton of such a system, including the code for a suitable scanner, is provided in an appendix to this paper. You need not modify the code that is already there; simply write out the parser functions. 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 would fail to recognize a correct sequence of Roman numbers, or would report success for an invalid sequence? Justify your answer. [2 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`.

4. In the compiler studied in the course the following Cocol code was used to handle code generation for a simple *IfStatement*:

```
IfStatement
=                    (. CGEN_labels testlabel; .)
   "if"
   "(" Condition ")"  (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Block              (. CGen->backpatch(testlabel); .) .
```

Suppose it was required to extend the system to provide an optional *else* clause:

```
IfStatement = "if" "(" Condition ")" Block [ "else" Block ] .
```

How would the Cocol specification have to be changed to generate efficient code for this extension? [6 marks]

This was covered in the practical course, although frequently inelegant solutions were proposed by students. One needs to be able to distinguish the two possibilities to the extent of not generating unnecessary branch instructions in the absence of the `else` clause:

```
IfStatement
=                          (. CGEN_labels testlabel, outlabel; .)
   "if" "(" Condition ")"  (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Block
   (   "else"              (. CGen->jump(outlabel, CGen->undefined);
                              CGen->backpatch(testlabel); .)
       Block               (. CGen->backpatch(outlabel); .)
     | /* no else part */   (. CGen->backpatch(testlabel); .)
   ) .
```

Alternatively, using a correction to the first "backpatch" should this prove necssary

```
IfStatement
=                          (. CGEN_labels testlabel, outlabel; .)
   "if" "(" Condition ")"  (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Block                   (. CGen->backpatch(testlabel); .)
   [   "else"              (. CGen->jump(outlabel, CGen->undefined);
                              CGen->backpatch(testlabel); .)
       Block               (. CGen->backpatch(outlabel); .)
   ] .
```

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

```
WhileStatement
=                       (. CGEN_labels whilelabel, testlabel, endlabel; .)
   "while"              (. CGen->storelabel(whilelabel); .)
   "(" Condition ")"    (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Block                (. CGen->jump(endlabel, whilelabel);
                           CGen->backpatch(testlabel); .) .
```

This generates code that matches the outline

```
whilelabel:   Condition
testlabel:    BZE  Continue
              Block
              BRN  whilelabel
continue:
```

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

```
whilelabel:   BRN  testlabel
looplabel:    Block
testlabel:    Condition
              BNZ  looplabel
continue:
```

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 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.

6. The following Cocol description is of a set of letters in envelopes ready to take to the post office.

```
COMPILER Mail $XCN
/* Describe simple set of mailing envelopes */

IGNORE CHR(1) .. CHR(13)

CHARACTERS
  eol       = CHR(13) .
  digit     = "0123456789" .
  inaddress = ANY - CHR(0) - '$:' - eol .
  sp        = CHR(32) .
TOKENS
  number    = "postcode:" sp digit { digit } .
  info      = inaddress { inaddress } .

PRODUCTIONS
  Mail      = { Envelope } EOF .
  Envelope  = Stamp { Stamp } Person Address .
  Stamp     = "$1" | "$2" | "$3" .          /* values of stamps implied */
  Address   = Street Town [ PostCode ] .
  Person    = info .
  Street    = info .
  Town      = info .
  PostCode  = number .
END Mail.
```

What would you have to add to this grammar so that the parser system could tell you (a) the total value of the stamps on all the envelopes (make the obvious assumption about the value of a stamp denoted by $x being the number x) (b) the names of all people whose addresses did **not** contain postcodes? For your (and Postman Pat's) convenience the grammar has been spread out on the last page of this question paper, which you may detach and submit with your answer book. [10 marks]

Two possible solutions follow:

```
COMPILER Mail $XCN
/* Describe simple set of mailing envelopes */
/* This does it all with "global" variables for simplicity */

#include <stdio.h>
int cost = 0;
char name[1000];

IGNORE CHR(1) .. CHR(13)

CHARACTERS
  eol       = CHR(13) .
  digit     = "0123456789" .
  inaddress = ANY - CHR(0) - '$:' - eol .
  sp        = CHR(32).

TOKENS
  number    = "postcode:" sp digit { digit } .
  info      = inaddress { inaddress } .

PRODUCTIONS
  Mail
  = { Envelope } EOF       (. printf("Total cost %d\n", cost); .) .

  Envelope
  = Stamp { Stamp } Person Address .

  Stamp
  =   "$1"                 (. cost += 1; .)
    | "$2"                 (. cost += 2; .)
    | "$3"                 (. cost += 3; .) .

  Address
  = Street Town
    (   PostCode
      |                    (. printf("%s has no postcode\n", name); .)
    ) .

  Person = info            (. LexString(name, 100); .) .
  Street = info .
  Town   = info .
  PostCode = number .
END Mail.


COMPILER Mail $XCN
/* Describe simple set of mailing envelopes */

#include <stdio.h>
int cost = 0;

IGNORE CHR(1) .. CHR(13)

CHARACTERS
  eol       = CHR(13) .
  digit     = "0123456789" .
  inaddress = ANY - CHR(0) - '$:' - eol .
  sp        = CHR(32).

TOKENS
  number    = "postcode:" sp digit { digit } .
  info      = inaddress { inaddress } .

PRODUCTIONS
  Mail
  =                    (. cost = 0; .)
    {
    Envelope
    } EOF              (. printf("Total cost %d\n", cost); .) .
```

```
  Envelope
  =                   (. char name[100];
                         bool hasCode; .)
    Stamp
    { Stamp
    } Person<name, 100>
    Address<hasCode>  (. if (!hasCode) printf("%s has no postcode\n", name); .)
  .

  Stamp
  =
      "$1"            (. cost += 1; .)
    | "$2"            (. cost += 2; .)
    | "$3"            (. cost += 3; .) .

  Address<bool &hasCode>
  = Street Town       (. hasCode = false; .)
    [ PostCode        (. hasCode = true; .)
    ] .

  Person<char * name, int length>
  = info              (. LexString(name, length - 1); .) .

  Street   = info .
  Town     = info .
  PostCode = number .
END Mail.
```

7. 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, list[10];
  read(item);
  while (item != 0) {
    if (item > 0) { total = total + item; }
    read(item);
  }
  println("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: [6 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 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 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.

8. (a) Briefly clarify what you understand by the terms "scope/visibility" and "existence" as they apply to the implementation of variables for block-structured imperative languages such as Parva, Java, C++, Pascal or Modula-2. [5 marks]

This is all strightforward 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.) [6 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.

```
void main () {
  int list[5], 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[3], last;         // compilation point 4
}
```

Point 1

Name	list	i	j	k							
Offset	1	6	7	8							

Point 2

Name	list	i	j	k	a	b	c				
Offset	1	6	7	8	9	10	11				

Point 3

Name	list	i	j	k	c	a	d				
Offset	1	6	7	8	9	10	11				

Point 4

Name	list	i	j	k	b	last					
Offset	1	6	7	8	9	12					

(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 or Java 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? [3 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.

Section B [90 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 EXAM.ZIP, modify any files that you like, and then copy all the files back to the blank diskette that will be provided.

During the translators course we have made frequent use of the Cocol/EBNF notation for expressing production rules, and in particular the use of the {} and [] meta-brackets introduced by Wirth in his 1977 paper. An example of a system of productions using this notation is as follows:

```
Home      = Family { Pets } [ Vehicle ] "house" .
Pets      = "dog" [ "cat" ] | "cat" .
Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
Family    = Parents { Children } .
Parents   = "Dad" "Mom" | "Mom" "Dad" .
Parents   = "Dad" | "Mom" .
Child     = "Margaret" | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
```

In analysing such productions and/or testing the grammar it has often been suggested that:

(a) they be rewritten without using the Wirth brackets, but using right recursive productions and an explicit ε, as for example

```
Home      = Family AllPets Transport "house" .
AllPets   = Pets AllPets | ε .
Transport = Vehicle | ε .
Pets      = "dog" PussyCat | "cat" .
PussyCat  = "cat" | ε .
Vehicle   = ( "scooter" | "bicycle" ) | "fourbyfour" .
Family    = Parents Offspring .
Offspring = Offspring Children | ε .
Parents   = "Dad" "Mom" | "Mom" "Dad" .
Parents   = "Dad" | "Mom" .
Child     = "Margaret" | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
```

(b) a check be made to see that all the non-terminals have been defined (this does not occur in the above case - `Children` is undefined);

(c) a check be made to see that each non-terminal is defined only once (this does not occur in the above case - there are two rules for `Parents`);

(d) a check be made to see that each non-terminal (other than the goal) must appear on the right side of at least one production (this does not occur in the above case; `Child` defines a non-teminal which does not appear in any other production).

As a useful service to others who might take this course in future years (and hopeful that, although you might encourage others to do so, you are not forced to do so yourself!), develop a system using Coco/R that will carry out the above transformations and checks.

Here are some suggestions for deriving a complete system:

(a) In the exam kit (`EXAM.ZIP`) you will find the executables and support files for Coco/R, as used in the practical course. There is also the skeleton of a grammar file `EBNF.ATG` with suitable definitions of character sets and tokens similar to those you have seen before.

(b) It should be obvious that you will need to set up a "symbol table". In the exam kit is supplied the skeleton of the support "include file" `EBNF.H` which has a rudimentary form of table that you might like to extend. The template list handler file `LIST.H` familiar from earlier practicals has also been supplied.

(c) Inventing additional non-terminal names (without clashing with others that might already exist) might be done with the aim of deriving, for example

```
Home = Family HomeSeq1 HomeOpt2 "house" .
```

(d) In the exam kit will be found some other example data and production sets to assist in your development of the system, and an executable derived from a model solution.

(e) After converting a set of productions from EBNF to BNF, try converting the BNF productions once again to check that your system works consistently.

```
COMPILER EBNF $XCN
/* Convert a set of EBNF productions to use BNF conventions, and carry out
   some rudimentary checks on their being properly defined.
   YOUR IDENTITY HERE */

#include "ebnf.h"

CHARACTERS
  cr       = CHR(10) .
  lf       = CHR(13) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_".
  digit    = "0123456789" .
  noquote1 = ANY - "'" - cr - lf .
  noquote2 = ANY - '"' - cr - lf .

IGNORE CHR(9) .. CHR(13)

COMMENTS FROM "(*" TO "*)"  NESTED

TOKENS
  nonterminal = letter {letter | lowline | digit} .
  terminal    = "'" noquote1 {noquote1} "'" | '"' noquote2 {noquote2} '"' .

PRODUCTIONS
  EBNF
  =                                   (. TABLE_InitTable(); .)
    { Production }
    EOF .

  Production
  =                                   (. char Name[100];
                                         int i; .)
    SYNC nonterminal                  (. LexString(Name, sizeof(Name) - 1);
                                         TABLE_Add(Name, i);
                                      .)
    WEAK "="                          /* obviously more needed here */
    SYNC "." .

END EBNF.
```

```
// Various common items for grammar handler

   #ifndef EBNF_H
   #define EBNF_H

   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   #include <stdarg.h>
   #include <ctype.h>
   #include <limits.h>

   #define  boolean  int
   #define  bool     int
   #define  true     1
   #define  false    0
   #define  TRUE     1
   #define  FALSE    0
   #define  maxint   INT_MAX

   #if __MSDOS__ || MSDOS || WIN32 || __WIN32__
   #  define  pathsep '\\'
   #else
   #  define  pathsep '/'
   #endif

   static void appendextension (char *oldstr, char *ext, char *newstr)
   // Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
   { int i;
     char old[256];
     strcpy(old, oldstr);
     i = strlen(old);
     while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
     if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
     if (ext[0] == '.') sprintf(newstr,"%s%s", old, ext);
       else sprintf(newstr, "%s.%s", old, ext);
   }

   typedef struct {
     char name[50];
     // you may need other members
   } ENTRIES;

   const int TABLE_MaxEntries = 1000;            // limit on table size
   static int TABLE_NEntries;                    // number of active entries
   static ENTRIES TABLE_Table[TABLE_MaxEntries]; // the table itself

   static void TABLE_InitTable() {
   // Initialise table
     TABLE_NEntries = 0;
   }

   static void TABLE_Add (char *name, int &position) {
   // Attempt to add name to the table, and return position as the
   // index where it was either added previously or added by this call
     if (TABLE_NEntries > TABLE_MaxEntries) {
       fprintf(stderr, "Table Overflow"); exit(1);
     }
     strcpy(TABLE_Table[TABLE_NEntries + 1].name, name); // sentinel
     position = 1;
     while (strcmp(name, TABLE_Table[position].name)) position++;
     if (position > TABLE_NEntries) TABLE_NEntries++;    // new entry
   }

   static void TABLE_ListProductions() {
   // List all entries to standard output file
     for (int i = 1; i <= TABLE_NEntries; i++)
       printf("%s\n", TABLE_Table[i].name);
     printf("\n");
   }
   #endif /* EBNF_H */
```

One solution to Part B is suggested below. This uses a statically declared array based table similar to those used in various of the practicals in this course, as suggested in the hints to the problem. A solution using the template `List` class could be developed on similar lines. The basic idea is quite simple - simply copy the text in the

productions to a list of strings stroring the tokens and metabrackets for each production. However, when the { RepeatedOption } or [SingleOption] construction is encountered, invent a name, store this in the string instead, and then start a stylised production for that name whose right hand side contains the RepeatedOption or SingleOption. The fact that the whole system is recursive handles the problem of options within options and so on effectively "for free".

The additions and modifications to the table handler on which it all depends are as follows:

```
// Various common items for grammar handler

#ifndef EBNF_H
#define EBNF_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define  boolean  int
#define  bool     int
#define  true     1
#define  false    0
#define  TRUE     1
#define  FALSE    0
#define  maxint   INT_MAX

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  define  pathsep '\\'
#else
#  define  pathsep '/'
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{ int i;
  char old[256];
  strcpy(old, oldstr);
  i = strlen(old);
  while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
  if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
  if (ext[0] == '.') sprintf(newstr,"%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

typedef enum { TABLE_LHS, TABLE_RHS, TABLE_BOTH } SIDES;

const int MaxName = 25;
const int MaxLen  = 100;

typedef struct {
  unsigned char name[MaxName];
  int left, right;
  unsigned char str[MaxLen][MaxName];
  int length;
} ENTRIES;

const int TABLE_MaxEntries = 1000;           // limit on table size
static int TABLE_NEntries;                   // number of active entries
static ENTRIES TABLE_Table[TABLE_MaxEntries]; // the table itself
static int TABLE_extraNames;

static void TABLE_MakeName (char *oldName, char *type, char *newName) {
  char ext[5];
  TABLE_extraNames++;
  strcpy(newName, oldName);
  strcat(newName, type);
  itoa(TABLE_extraNames, ext, 10);
  strcat(newName, ext);
}

static void TABLE_InitTable() {
// Initialise table
  TABLE_NEntries = 0;
  TABLE_extraNames = 0;
}
```

```
static void TABLE_Add (char *name, SIDES where, int &position) {
// Add a nonterminal name name to the table, according as to where in a
// production it was found, and return position as the
// index where it was either added previously or added by this call
// debug:  printf("add %s\n", name);
  if (TABLE_NEntries == TABLE_MaxEntries) {
    fprintf(stderr, "Table Overflow 1"); exit(1);
  }
  strcpy(TABLE_Table[TABLE_NEntries + 1].name, name); // sentinel
  TABLE_Table[TABLE_NEntries + 1].left = 0;
  TABLE_Table[TABLE_NEntries + 1].right = 0;
  TABLE_Table[TABLE_NEntries + 1].length = 0;
  position = 1;
  while (strcmp(name, TABLE_Table[position].name)) position++;
  if (position > TABLE_NEntries) TABLE_NEntries++; // new entry
  switch (where) {
    case TABLE_LHS  :
      TABLE_Table[position].left++; break;
    case TABLE_RHS  :
      TABLE_Table[position].right++; break;
    case TABLE_BOTH :
      TABLE_Table[position].left++;
      TABLE_Table[position].right++; break;
  }
}

static void TABLE_AddText (char *Text, int i) {
// Add Text to the list of strings for the rule in the table at position i
// debug: printf("addtext %s\n", Text);
  strcpy(TABLE_Table[i].str[TABLE_Table[i].length], Text);
  TABLE_Table[i].length++;
  if (TABLE_Table[i].length == MaxLen) {
    fprintf(stderr, "Table Overflow 2"); exit(1);
  }
}

static void TABLE_ListProductions() {
// List all productions to standard output file
  for (int i = 1; i <= TABLE_NEntries; i++)
    if (TABLE_Table[i].length > 0) {
      int paren = 0;
      for (int j = 0; j < TABLE_Table[i].length; j++) {
        if (strcmp("(", TABLE_Table[i].str[j]) == 0) paren++;
        if (strcmp(")", TABLE_Table[i].str[j]) == 0) paren--;
        if (strcmp("|", TABLE_Table[i].str[j]) == 0 && paren == 0) // neater output for alternatives
          printf("\n%*c ", strlen(TABLE_Table[i].str[0]) + 1, ' ');
        printf(" %s", TABLE_Table[i].str[j]);
      }
    }
  printf("\n");
}

static void TABLE_TestProductions() {
// Check that all non terminals have appeared once on the left side of
// each production, and at least once on the right side of each production
  bool OK = TRUE; // optimistic
  int i;
  for (i = 1; i <= TABLE_NEntries; i++) {
    if (TABLE_Table[i].left == 0) {
      OK = false;
      printf("(* %s never defined *)\n", TABLE_Table[i].name);
    }
    else if (TABLE_Table[i].left > 1) {
      OK = false;
      printf("(* %s defined more than once *)\n", TABLE_Table[i].name);
    }
  }
  for (i = 2; i <= TABLE_NEntries; i++) {
    if (TABLE_Table[i].right == 0) {
      OK = false;
      printf("(* %s cannot be reached *)\n", TABLE_Table[i].name);
    }
  }
  if (!OK) printf("\n(* Cannot be correct grammar *)\n");
}

#endif /* EBNF_H */
```

```
COMPILER Ebnf $XCN
/* Convert a set of EBNF productions to use BNF conventions, and carry out
   some rudimentary checks on their being properly defined. */

#include "ebnf.h"

CHARACTERS
  cr       = CHR(10) .
  lf       = CHR(13) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_".
  digit    = "0123456789" .
  noquote1 = ANY - "'" - cr - lf .
  noquote2 = ANY - '"' - cr - lf .

IGNORE CHR(9) .. CHR(13)

COMMENTS FROM "(*" TO "*)"  NESTED

TOKENS
  nonterminal = letter {letter | lowline | digit} .
  terminal    = "'" noquote1 {noquote1} "'" | '"' noquote2 {noquote2} '"' .

PRODUCTIONS
  Ebnf
  =                                   (. TABLE_InitTable(); .)
    { Production }
    EOF                               (. if (Successful()) {
                                           TABLE_ListProductions();
                                           TABLE_TestProductions();
                                         }
                                      .)
    .

  Production
  =                                   (. char Name[MaxName];
                                         int i; .)
    SYNC nonterminal                  (. LexString(Name, sizeof(Name) - 1);
                                         TABLE_Add(Name, TABLE_LHS, i);
                                         TABLE_AddText(Name, i);
                                         TABLE_AddText("= ", i);
                                      .)
    WEAK "=" Expression<Name, i>
    SYNC "."                          (. TABLE_AddText(".\n", i); .)
    .

  Expression<char *s, int i>
  = Term<s, i> { WEAK "|"             (. TABLE_AddText("|", i); .)
    Term<s, i> } .

  Term<char *s, int i> = [ Factor<s, i> { Factor<s, i> } ] .

  Factor<char *s, int i>
  =                                   (. char Name[MaxName];
                                         int j .)
      nonterminal                     (. LexString(Name, sizeof(Name) - 1);
                                         TABLE_Add(Name, TABLE_RHS, j);
                                         TABLE_AddText(Name, i);
                                      .)
    | terminal                        (. LexString(Name, sizeof(Name) - 1);
                                         TABLE_AddText(Name, i);
                                      .)
    | "["                             (. TABLE_MakeName(s, "Opt", Name);
                                         TABLE_AddText(Name, i);
                                         TABLE_Add(Name, TABLE_BOTH, j);
                                         TABLE_AddText(Name, j);
                                         TABLE_AddText("= ", j); .)
      Expression<s, j>
      "]"                             (. TABLE_AddText("|", j);
                                         TABLE_AddText("ε .\n", j); .)
    | "("                             (. TABLE_AddText("(", i); .)
      Expression<s, i>
      ")"                             (. TABLE_AddText(")", i); .)
    | "{"                             (. TABLE_MakeName(s, "Seq", Name);
                                         TABLE_AddText(Name, i);
                                         TABLE_Add(Name, TABLE_BOTH, j);
                                         TABLE_AddText(Name, j);
                                         TABLE_AddText("= ", j);
                                         TABLE_AddText("(", j); .)
      Expression<s, j>                (. TABLE_AddText(")", j);
                                         TABLE_AddText(Name, j);
```

```
                                          TABLE_AddText("|", j);
                                          TABLE_AddText("ε .\n", j);
                                        .)
        "}"
      | "ε"                             (. TABLE_AddText("ε", i); .)
      .

END Ebnf.
```

Another solution might be as follows. Here we have taken cognisance of the fact that there is nothing really wrong with productions like

```
Supper =  "pizza" .
Supper =  "lasagna" .
```

which are of course equivalent to

```
Supper = "pizza" | "lasagna" .
```

so that if a further rule is discovered for a nonterminal we can simply concatenate the alternatives together.

The tablehandler needs some changes (only the changes are shown here)

```
static void TABLE_ListProductions() {
// List all productions to standard output file
  for (int i = 1; i <= TABLE_NEntries; i++)
    if (TABLE_Table[i].length > 0) {
      int paren = 0;
      for (int j = 0; j < TABLE_Table[i].length; j++) {
        if (strcmp("(", TABLE_Table[i].str[j]) == 0) paren++;
        if (strcmp(")", TABLE_Table[i].str[j]) == 0) paren--;
        if (strcmp("|", TABLE_Table[i].str[j]) == 0 && paren == 0) // neater output for alternatives
          printf("\n%*c ", strlen(TABLE_Table[i].str[0]) + 1, ' ');
        printf(" %s", TABLE_Table[i].str[j]);
      }
      printf(" .\n");        // ================== added
    }
  printf("\n");
}

static int TABLE_PositionLHS (char *name) {
// Return the position in the table where name can be found on the LHS,
// or 0 if not yet entered found in that way
// debug: printf("positionLHS %s %d\n", name, TABLE_NEntries);
  if (TABLE_NEntries > TABLE_MaxEntries) {
    fprintf(stderr, "Table Overflow"); exit(1);
  }
  strcpy(TABLE_Table[0].name, name); // sentinel
  int position = TABLE_NEntries;
  while (strcmp(name, TABLE_Table[position].name)) position--;
  if (position > 0 && TABLE_Table[position].left != 0) return position;
  else return 0;
}

static void TABLE_DefinedAgain (int i) {
// Marks entry i as having had LHS defined more than once
  TABLE_Table[i].left++;
}
```

The grammar also needs a few changes:

```
      Production
      =                                 (. char Name[MaxName];
                                           int i; .)
        SYNC nonterminal                (. LexString(Name, sizeof(Name) - 1);
***                                        i = TABLE_PositionLHS(Name);
***                                        if (i == 0) {
***                                          TABLE_Add(Name, TABLE_LHS, i);
***                                          TABLE_AddText(Name, i);
***                                          TABLE_AddText("= ", i);
***                                        } else {
***                                          TABLE_AddText("|", i);
***                                          TABLE_DefinedAgain(i);
***                                        }
                                         .)
        WEAK "=" Expression<Name, i>
```

```
***     SYNC "."
        .

        Factor<char *s, int i>
        =                                   (. char Name[MaxName];
                                               int j .)
            nonterminal                     (. LexString(Name, sizeof(Name) - 1);
                                               TABLE_Add(Name, TABLE_RHS, j);
                                               TABLE_AddText(Name, i);
                                            .)
          | terminal                        (. LexString(Name, sizeof(Name) - 1);
                                               TABLE_AddText(Name, i);
                                            .)
          | "["                             (. TABLE_MakeName(s, "Opt", Name);
                                               TABLE_AddText(Name, i);
                                               TABLE_Add(Name, TABLE_BOTH, j);
                                               TABLE_AddText(Name, j);
                                               TABLE_AddText("= ", j); .)
            Expression<s, j>
            "]"                             (. TABLE_AddText("|", j);
***                                            TABLE_AddText("ε", j); .)
          | "("                             (. TABLE_AddText("(", i); .)
            Expression<s, i>
            ")"                             (. TABLE_AddText(")", i); .)
          | "{"                             (. TABLE_MakeName(s, "Seq", Name);
                                               TABLE_AddText(Name, i);
                                               TABLE_Add(Name, TABLE_BOTH, j);
                                               TABLE_AddText(Name, j);
                                               TABLE_AddText("= ", j); .)
            Expression<s, j>                (. TABLE_AddText(Name, j);
                                               TABLE_AddText("|", j);
***                                            TABLE_AddText("ε", j);
                                            .)
            "}"
          | "ε"                             (. TABLE_AddText("ε", i); .)
          .
```