RHODES UNIVERSITY

November Examinations - 2000 - Solutions

Computer Science 301 - Paper 1

Examiners:	Time 3 hours
Prof P.D. Terry	Marks 180
Prof D. Kourie	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, and 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:

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

   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

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

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

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

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 $A \ \varepsilon\ N$

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.

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]*

An ambiguous grammar is one for which there is at least one string in the language that can be derived in more than one way from the goal symbol using the given set of productions.

It is quite easy to write down ambiguous grammars. The obvious example to quote is the "dangling else" one, though it is hard to give an equivalent unambiguous one. An easier example would be the one we discussed in class for Roman numerals:

```
Ambiguous:

    Units =    [ "I" ] [ "I" ] [ "I" ]
             | "IV" | "IX"
             | "V" [ "I" ] [ "I" ] [ "I" ]

Unambiguous:

    Units =    [ "I" [ "I" [ "I" ] ] ]
             | "IV" | "IX"
             | "V" [ "I" [ "I" [ "I" ] ] ]
```

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 .
```

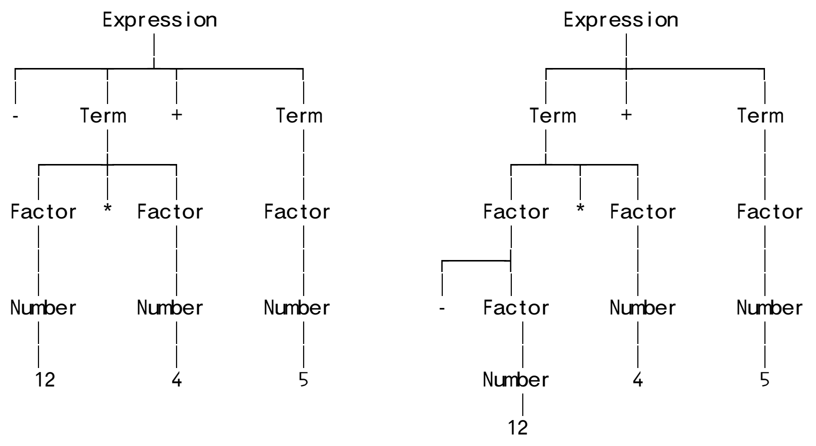
while the other suggested that the correct answer would be (call this GRAMMAR B)

```
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]*

Both grammars are correct, in the sense that they can derive these sorts of strings with the correct precedence attached to the operators. The leading unary minus in the first grammar will be bound to the first term only, and because (-a)*b = -(a*b) = a*(-b) the arithmetic meaning would be retained. In the second grammar the meaning of -a*b would be explicitly (-a)*b, which is the same. (Quite unbelievably, several students claimed that this would not be the case. Where, one wonders, did they learn arithmetic!) The subtle difference is that the second grammar would allow expressions like a * - b or a + - b or even a - - - - b, which the first will not do. An expression like a - - b would be taken to mean a - (-b) = a + b, so all would be well. It is of interest to record that C++ incorporates the ideas of the second grammar, while Pascal/Modula incorporate the first ideas.

The parse trees are as follows



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);
```

We add a (very) simple `SwitchStatement` to the statement types, and allow statements to be prefixed with `Labels`. All that is needed is this:

```
Block = "{" { Statement } "}" .
Statement
= { Label }                                  // ================
  (  Block            | Assignment
   | IfStatement      | WhileStatement
   | ForStatement     | RepeatStatement
   | ReadStatement    | WriteStatement
   | ReturnStatement  | EmptyStatement
   | ConstDeclaration | VarDeclarations
   | SwitchStatement  | BreakStatement ) .   // ================

Label = "case" ConstExpression ":" | "default" ":" .
SwitchStatement = "switch" "(" Expression ")" Statement .
ConstExpression = Expression .
BreakStatement = "break" ";" .
```

(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]*

As you should realise from looking at the simple grammar hard enough, the `switch` statement in C++ is an absolute monstrosity. It is really a collection of unstructured "goto" statements. However, a compiler would have to impose some form of constraints. In particular:

- the type of the `Expression` in the selector portion and of each `ConstExpression` in the labels in the scope of the associated `Statement` should all be identical. Actually in C++ more or less anything is compatible with more or less anything else of course, so this would probably not matter. But we could insist on it for Topsy!
- the value of each `ConstExpression` in the scope of this associated statement should be unique.
- there should not be more than one `default` label in the scope of this associated statement.
- the expressions used in the labels must be ones that yield constant values computable at compile time (as implied by the names of the non-terminals above).

So the following might be illegal in several respects:

```
int i;
switch (i) {
  default : i = 9;
  case 1 : case true : i = 4;     // label 1 used twice, true is not int
  case 1 : case i : case 'y' : i = 5; break; // 'y' is not int; i is variable
  default : i = 10;               // only one default allowed
}
```

While it would be nice to insist that the `SwitchStatement` is invalid unless there is a matching label for every possible value of the selector `Expression` (as is required in Modula-2, for example), this is not required in C++. Nor is it required to have `Break` statements in each option, though it is often highly necessary to do so. And while a statement like

```
switch(4) ;
```

is quite hopelessly useless, and one like

```
switch (3) {
  default : x = y;    // no need for default to be last
  case 1: a = b;      // could never get here
  case 2: c = d;      // could never get here
  case 3: e = f;
}
```

looks silly, they are both legal.

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]

We might do worse than to rewrite these without the metabrackets:

```
PRAC      = Handouts Tasks WriteUp "Submit" .
Tasks     = Task Tasks | ε .
WriteUp   = Essay | ε .
Handouts = "PracSheet" Hints
Hints     = "HintSheet" Hints | ε .
Task      = "Attempt" TryAgain .
TryAgain = Help  "Attempt" TryAgain | ε  .
Essay     = "CorelFormat" | "MSWordFormat" | "ASCIIFormat" .
Help      = Tutor Lecturer .
Tutor     = "Pat" | "Holger" | "Barry" | "Colin" | "Guy" | ε .
Lecturer = "Pat" | ε .
```

Clearly all the productions satisfy Rule 1. There are several nullable non-terminals. Checking Rule 2 for each of these:

```
FIRST(Lecturer) = "Pat"
FIRST(Tutor)    = "Pat"  "Holger"  "Barry"  "Colin"  "Guy"
FIRST(Help)     = "Pat"  "Holger"  "Barry"  "Colin"  "Guy"
FIRST(TryAgain) = "Attempt"  "Pat"  "Holger"  "Barry"  "Colin"  "Guy"
FIRST(Hints)    = "HintSheet"
FIRST(WriteUp)  = "CorelFormat"  "MSWordFormat"  "ASCIIFormat"
FIRST(Tasks)    = "Attempt"

FOLLOW(Lecturer) = "Attempt"
FOLLOW(Tutor)    = "Attempt"  "Pat"
FOLLOW(Help)     = "Attempt"
FOLLOW(TryAgain) = "Submit"  "Attempt"  "CorelFormat"  "MSWordFormat"  "ASCIIFormat"
FOLLOW(Hints)    = "Submit"  "Attempt"  "CorelFormat"  "MSWordFormat"  "ASCIIFormat"
FOLLOW(WriteUp)  = "Submit"
FOLLOW(Tasks)    = "Submit"  "CorelFormat"  "MSWordFormat"  "ASCIIFormat"
```

From which we see that Rule 2 is broken for `Tutor` and for `TryAgain`. Most students saw that the rule for `Tutor` was broken, rather fewer did the analysis or saw the problem for `TryAgain`. However, I was pleased to see that most people could rewrite productions without meta brackets.

This is easily derived by making a submission to Coco:

```
COMPILER PRAC $TF
PRODUCTIONS
  PRAC     = Handouts Tasks WriteUp "Submit" .
  Tasks    = Task Tasks |   .
  WriteUp  = Essay |   .
  Handouts = "PracSheet" Hints .
  Hints    = "HintSheet" Hints |   .
  Task     = "Attempt" TryAgain .
  TryAgain = [ Help  "Attempt" TryAgain ]  .
  Essay    = "CorelFormat" | "MSWordFormat" | "ASCIIFormat" .
  Help     = Tutor Lecturer .
  Tutor    = [ "Pat" | "Holger" | "Barry" | "Colin" | "Guy" ] .
  Lecturer = [ "Pat" ]   .
END PRAC.
```

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]*

What was required here was a discussion of the material in Chapter 10, in particular the section:

```
IF Sym   FIRST(A) THEN
  ReportError; SkipTo(FIRST(A) U FOLLOW(A))
END;
IF SYM.Sym   FIRST(A) THEN
  Parse(A);
  IF SYM.Sym   FOLLOW(A) THEN
    ReportError; SkipTo(FOLLOW(A))
  END
END
```

Although the FOLLOW set for a non-terminal is quite easy to determine, the legitimate follower may itself have been omitted, and this may lead to too many symbols being skipped at routine exit. To prevent this, a parser using this approach usually passes to each sub-parser a *Followers* parameter, which is constructed so as to include

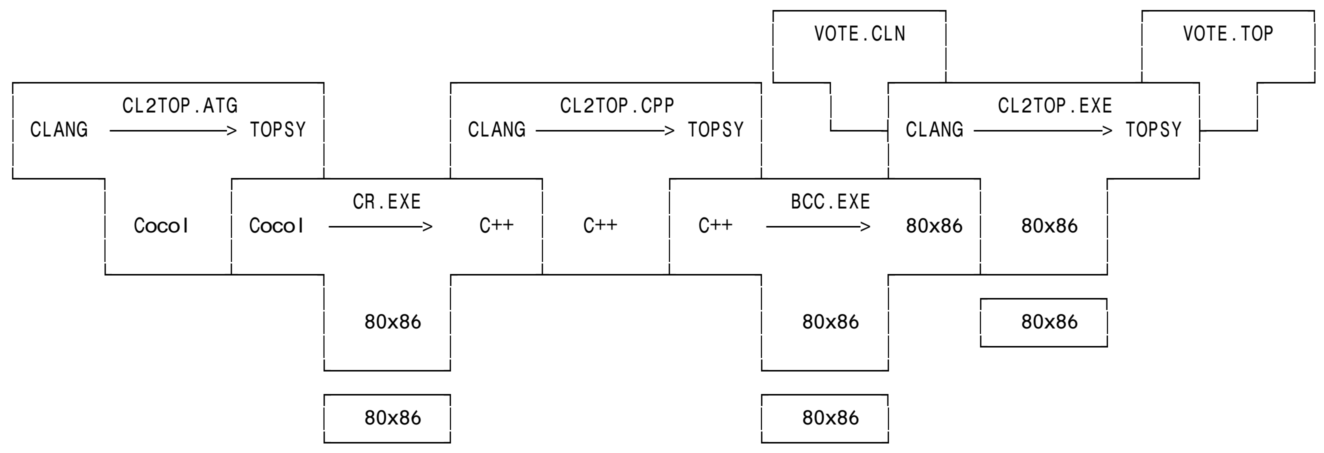
- the minimally correct set FOLLOW(*A*), augmented by
- symbols that have already been passed as *Followers* to the calling routine (that is, later followers), and also
- so-called **beacon symbols**, which are on no account to be passed over, even though their presence would be quite out of context. In this way the parser can often avoid skipping large sections of possibly important code.

On return from sub-parser *A* we can then be fairly certain that *Sym* contains a terminal which was either expected (if it is in FOLLOW(*A*)), or can be used to regain synchronization (if it is one of the beacons, or is in FOLLOW(*Caller(A)*)). The caller may need to make a further test to see which of these conditions has arisen.

Many students gave answers in terms of the `SYNC` and `WEAK` keywords of Coco/R, which I handled as generously as possible!

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]*

   Hurrah! AT LAST nearly everyone got this correct!



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]

It's the action that counts! Simply relax the syntax to allow any number of W, X, Y, Z to appear in any order; count how many of each there are, and then check afterwards that each has appeared exactly once:

```
A
=                  (. int W = 0, X = 0, Y = 0, Z = 0; .)
  {  W             (. W++; .)
   | X             (. X++; .)
   | Y             (. Y++; .)
   | Z             (. Z++; .)
  }                (. if (W * X * Y * Z != 1) SemError(1000); .) .
```

Sadly, virtually nobody came up with this (though there were some complicated solutions that used actions to build up tables and so on. There were also a few students who came up with solutions where they tried recording strings, but note that the problem was posed in terms of *non terminals* W, X, Y and Z, so that `LexString` would have been of no use really. Ah well. I keep trying to tell people that behind every problem there is a simple solution waiting to be discovered. There really is!

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
    Staff[1] = rich;       // Invalid assignment - type incompatibility
  Manager = Boss;          // Cannot assign to a constant
  PayPacket[Boss] = 1000; // Incompatible subscript type
}
```

As usual, I have provided a kit with the full sources for my solution on the course WWW pages in Pascal and C++ versions. There must be several ways of solving the problem posed; this is the one that occurred to me. As always, I try to illustrate a simple approach, which, sadly only a very few people seemed to have thought of for themselves. At the heart of this is the idea that the concept of `type` is modelled in the Topsy system by enumerating the types themselves as a set of small integers (I tried to thrust this at you in the question itself, in a sense). Each type has a unique internal "number" assigned to it - all we have to do is extend the range of these numbers and virtually everything else was already provided in the compiler for you already! I guess very few people had really studied the code of the original Topsy ATG file in anything like the detail I had hoped.

The syntactic changes we need to solve the problem of adding enumeration types to Topsy are simple, and most people at least got the essential grammar for this correct. We need to extend the syntax for `VarDeclarations`, which might conveniently introduce a new non-terminal `EnumType`:

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

EnumType = "enum" Ident "{" OneEnum { "," OneEnum } "}"

OneEnum = Ident .
```

Essentially all the `EnumType` parser will have to do is inject a whole lot of identifiers into the symbol table, most of them being constants with successive small integral values, generated automatically as the parse proceeds.

Many people either did not understand casting, or else did not get round to trying it out. Many of those that did were terribly restrictive, and several of these totally confused what happens at compile time with what should happen at run time. Syntactically, to achieve type casting or type transfer we simply need to extend `Factor`, in a general a way as possible (not, as some did, as a special kind of assignment statement).

Two ways of achieving the conversion "to" an enumerated type are shown, one using a `Designator` followed by an `Expression` in parentheses, which requires the parsing to be driven semantically, as will be seen later, and the other using the key word "`val`" as is used in Modula-2, where the effect is actually easier to achieve:

```
Factor
=    Designator [ "(" Expression ")" ]
   | Number | Char | "true" | "false"
   | ( "int" | "char" | "bool" ) "(" Expression ")"
   | "val" "(" Ident "," Expression ")"
   | "(" Expression ")" .
```

As already mentioned, the secret in solving this problem neatly lies in finding a simple way to add types to the language. In the original system the four basic types were themselves defined by an enumeration. In an extended system we might replace this enumeration with a set of explicit simple integers. The header for the table class is altered to contain the declarations:

```
typedef int TABLE_types;   // no longer enumeration

const int TABLE_none  = 0; // basic types - same intrinsic values as before
const int TABLE_ints  = 1;
const int TABLE_chars = 2;
const int TABLE_bools = 3;
```

Of course, in C++ we can actually leave the original `typedef` for `TABLE_types` in place, because in C++ we can go beyond the end of an enumeration later on. You could not have done this in Pascal, and so I have illustrated the clearer way that works for both.

We need to extend the `TABLE_entries` structure to allow for a new class of identifier - `TABLE_enums`:

```
enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs, TABLE_enums };
```

Several people clearly had not seen the important distinction between the `idclass` and `type` concept, and tried adding a `TABLE_enum` item to the `TABLE_type` enumeration.

At the heart of the enumeration type concept is the need to have some way of recording the maximum value that a variable can assume, so that range checking can be implemented. This becomes a property of each variable and also of the enumeration "type" itself. In the original system this was not necessary - the only limited range was for characters, and this was handled in an ad-hoc way. I had hoped that drawing attention to this in several questions in Task 2 of Practical 25 would have alerted people to what was needed. For simplicity all types should have this new attribute, of course:

```
struct TABLE_entries {
  TABLE_alfa name;            // identifier
  TABLE_idclasses idclass;    // class
  int self;
  TABLE_types type;
  union {
    struct {
      int value;
    } c;                      // constants
    struct {
      int size, offset, max;  // -------  modified
      bool canchange, scalar;
    } v;                      // variables
    struct {                  // added
      int maxValue;
    } e;                      // -------  enumerations
  };
};
```

In my solution I found it convenient to extend the `TABLE::printtable` method for testing purposes:

```
void TABLE::printtable(FILE *lst)
{ putc('\n', lst);
  for (int i = 1; i <= top; i++)
  { fprintf(lst, "%5d %-16s", i, symtable[i].name);
    fprintf(lst, "%3d", symtable[i].type);
    switch (symtable[i].idclass)
```

```
      { case TABLE_consts:
          fprintf(lst, " Constant %6d\n", symtable[i].c.value); break;
        case TABLE_vars:
          fprintf(lst, " Variable %6d %6d %6d\n", symtable[i].v.offset,
                      symtable[i].v.size, symtable[i].v.max); break;
        case TABLE_progs:
          fprintf(lst, " Program\n"); break;
        case TABLE_enums:
          fprintf(lst, " EnumType %6d\n", symtable[i].e.maxValue); break;
      }
    }
  }
```

Incidentally, many solutions showed that students did not understand the `union` construction at all, as they assumed all the "fields" were available regardless of which `idclass` was under consideration.

To illustrate how we store all this contextual information, here is a sample program like `T2.TOP` and the symbol table it produced:

```
    void main (void) { // Several enumerations
      enum MyType { a, b, c, d } X, Y, Z[4];
      enum YourType { p, q } A;
      enum SillyType { LonelyValue } Ace;
    }

      Name         type idclass     value   offset  size  max

     1 a              4 Constant      0
     2 b              4 Constant      1
     3 c              4 Constant      2
     4 d              4 Constant      3
     5 MyType         4 EnumType                             3
     6 X              4 Variable                1     1      1
     7 Y              4 Variable                2     1      1
     8 Z              4 Variable                3     4      1
     9 p              5 Constant      0
    10 q              5 Constant      1
    11 YourType       5 EnumType                             1
    12 A              5 Variable                7     1      1
    13 LonelyValue    6 Constant      0
    14 SillyType      6 EnumType                             0
    15 Ace            6 Variable                8     1      0
```

Range checking in our virtual machine, as hopefully everyone should have known after studying Practicals 25 and 26, can be achieved by the use of special versions of `PPP`, `MMM`, `STO` and `POP`. In the original system this aspect of code generation was driven by a Boolean parameter. In an extended system one can pass the maximum value to the code generation routines explicitly. Note that integers are treated as a special case, as we do not usually worry about range checking for these.

```
  void CGEN::increment(int maxValue)
  { if (maxValue != maxint) { emit(int(STKMC_ppr)); emit(maxValue); } else emit(int(STKMC_ppp)); }

  void CGEN::decrement(int maxValue)
  { if (maxValue != maxint) { emit(int(STKMC_mmr)); emit(maxValue); }  else emit(int(STKMC_mmm)); }

  void CGEN::assign(int maxValue)
  { if (maxValue != maxint) { emit(int(STKMC_str)); emit(maxValue); }  else emit(int(STKMC_sto)); }

  void CGEN::store(int offset, int maxValue)
  { if (maxValue != maxint) { emit(int(STKMC_lit)); emit(maxValue); emit(int(STKMC_por)); }
    else emit(int(STKMC_pop));
    emit(-offset);
  }
```

With that background, let us examine the modifications needed to the attributed grammar:

Firstly we need to be able to extend the "range" of the `typeset` type, as we need to have type numbers potentially reaching fairly large values. 255 is a value for demonstration purposes only.

```
  typedef Set<255> typeset;                     /* ------------- extended */
```

Each successive new enumeration type is then distinguished by the next value in a sequence which starts at 0 (for

TABLE_none) and progresses through the values 1, 2 and 3 to TABLE_bool It is convenient to have a global integer to record the last type introduced in this way (a neater solution would hide this in the table handler module, but I was lazy). Initially this is set to TABLE_bools just before parsing commences. It is also convenient to have a set type similar to arithtypes and booltypes that can be used to accumulate the types suitable for controlling for loops and for which the ++ and -- operations are permissible in other contexts as well:

```
int lastType;                                /* ------------- extended */
typeset controltypes, arithtypes, booltypes;  /* ------------- extended */
```

Compatibility has to be *very* strict for enumerations. As it happens, the essence of type checking can *all* be achieved by a very simple modification to the compatible function:

```
bool compatible (TABLE_types atype, TABLE_types btype)
// returns true if atype is compatible with btype
{ return arithtypes.memb(atype) && arithtypes.memb(btype)
         || booltypes.memb(atype) && booltypes.memb(btype)
         || atype == btype;                  /* ------------- extended */
}
```

The VarDeclaration parser uses a local variable max to record the largest internal value a variable may be assigned. This is then passed to the OneVar parser so that it may be recorded in the appropriate symbol table entry. Note that for integers we use the value of maxint, which can be detected as a special case by the code generator, as you will already have seen.

```
VarDeclarations<int &framesize>            /* ------------- extended */
=                               (. TABLE_types type; int max; .)
   (   "int"                    (. type = TABLE_ints; max = maxint; .)
     | "char"                   (. type = TABLE_chars; max = 255; .)
     | "bool"                   (. type = TABLE_bools; max = 1; .)
     | EnumType<type, max>                 /* ------------- extended */
   )
   OneVar<framesize, type, max> { WEAK "," OneVar<framesize, type, max> } WEAK ";" .
```

The EnumType parser must return a maximum value for the type it defines and introduces, and must itself make an entry into the symbol table for the associated type name, so that this may later be recognisable if and when it is used in the role of a casting function name. Note that this parser increments the lastType counter to define the effective type information for each of the constants and variables that are about to be introduced. This type is also added to the controltypes set.

```
EnumType<TABLE_types &type, int &max>      /* ------------- extended */
=                               (. int value;
                                   TABLE_entries entry; .)
   "enum" Ident<entry.name>     (. entry.idclass = TABLE_enums;
                                   lastType++; type = lastType;
                                   entry.type = lastType;
                                   controltypes.incl(lastType);
                                   max = -1; .)
   "{" OneEnum<max, type>
       { "," OneEnum<max, type> }
   "}"                          (. entry.e.maxValue = max;
                                   Table->enter(entry); .) .
```

The OneEnum parser is very simple. It makes an entry for a constant with the associated (inherited) type into the symbol table. As a side effect it is responsible for incrementing the value for each successive constant automatically:

```
OneEnum<int &value, TABLE_types type>
=                               (. TABLE_entries entry; .)
   Ident<entry.name>            (. value++; entry.c.value = value;
                                   entry.idclass = TABLE_consts;
                                   entry.type = type;
                                   Table->enter(entry); .) .
```

The OneVar parser is much as before, except that it needs to record the maximum value that a variable can assume. Note that if a variable is initialised then its maximum value is passed to the CGen->store method so that the correct range checking can be effected. Many solutions submitted introduced a new production almost identical to OneVar, but this really is totally unnecessary. Don't be beguiled by the ease with which one can cut and paste in editors!

```
OneVar<int &framesize, TABLE_types type, int max>
=                              (. TABLE_entries entry;
                                  TABLE_types exptype; .)
                               (. entry.idclass = TABLE_vars; entry.v.size = 1;
                                  entry.v.scalar = true; entry.type = type;
                                            /* ------------ extended */
                                  entry.v.max = max;
                                  entry.v.offset = framesize + 1; .)
   Ident<entry.name>
   [ ArraySize<entry.v.size>   (. entry.v.scalar = false; .)
   ]                           (. Table->enter(entry);
                                  framesize += entry.v.size;
                                  if (framesize > maxframe) maxframe = framesize; .)
   [ AssignOp                  (. if (!entry.v.scalar) SemError(210); .)
     Expression<exptype>       (. if (!compatible(entry.type, exptype)) SemError(218);
                                            /* ------------ extended */
                                  else CGen->store(entry.v.offset, entry.v.max); .)
   ] .
```

The `Assignment` parser also has to be sensitive to the modified `CGen->store` and `CGen->assign` methods:

```
Assignment
=                              (. TABLE_types vartype, exptype;
                                  TABLE_entries entry; .)
   Variable<vartype, entry>
   ( AssignOp
     Expression<exptype>       (. if (compatible(vartype, exptype))
                                            /* ------------ extended */
                                    if (entry.v.scalar) CGen->store(entry.v.offset, entry.v.max);
                                    else CGen->assign(entry.v.max);
                                  else SemError(218); .)
     | "++"                    (. if (entry.v.scalar) CGen->stackaddress(entry.v.offset);
                                  if (controltypes.memb(vartype))
                                            /* ------------ extended */
                                    CGen->increment(entry.v.max);
                                    else SemError(218); .)
     | "--"                    (. if (entry.v.scalar) CGen->stackaddress(entry.v.offset);
                                  if (controltypes.memb(vartype))
                                            /* ------------ extended */
                                    CGen->decrement(entry.v.max);
                                    else SemError(218); .)
   ) WEAK ";" .
```

`For` statements are quite tricky to handle. It is appropriate to allow them to be controlled by enumerated types, but I decided to abandon the range checking in the epilogue for reasons which you might like to puzzle out for yourselves. Virtually nobody made any attempt to deal with `for` statements properly!

```
ForStatement<int &framesize>
=                              (. CGEN_labels startloop, testlabel, bodylabel,
                                              startepilogue, oldexit, dummylabel;
                                  TABLE_types controltype, exptype;
                                  TABLE_entries entry1, entry2; .)
   "for" WEAK "("
   Designator<classset(TABLE_vars), entry1, controltype>
                                            /* ------------ extended */
                               (. if (!entry1.v.scalar || !controltypes.memb(controltype))
                                    SemError(225);
                                  if (!entry1.v.canchange) SemError(229); .)
   AssignOp
   Expression<exptype>         (. if (compatible(controltype, exptype))
                                            /* ------------ extended */
                                    CGen->store(entry1.v.offset, entry1.v.max);
                                    else SemError(218);
                                  CGen->storelabel(startloop) .)
   WEAK ";" Condition          (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   [ ";"                       (. CGen->jump(bodylabel, CGen->undefined);
                                  CGen->storelabel(startepilogue); .)
      Designator<classset(TABLE_vars), entry2, controltype>
                               (. if (strcmp(entry1.name, entry2.name))
                                    SemError(226); .)
      (   AssignOp
          Expression<exptype> (. if (compatible(controltype, exptype))
                                            /* ------------ extended */
                                    CGen->store(entry2.v.offset, maxint);
                                  else SemError(218); .)
        | "++"                 (. CGen->stackaddress(entry2.v.offset);
                                          /* ------------- extended */
                                 CGen->increment(maxint); .)
          | "--"              (. CGen->stackaddress(entry2.v.offset);
                                          /* ------------- extended */
                                 CGen->decrement(maxint); .)
        )                     (. CGen->jump(dummylabel, startloop);
                                 CGen->backpatch(bodylabel);
                                 startloop = startepilogue; .)
      ]
      ")"                     (. looplevel++; oldexit = loopexit;
                                 loopexit = CGen->undefined;
                                 Table->protect(entry1) .)
      Statement<framesize>    (. CGen->jump(dummylabel, startloop);
                                 CGen->backpatch(testlabel);
                                 CGen->backpatchlist(loopexit);
                                 loopexit = oldexit; looplevel--;
                                 Table->unprotect(entry1); .) .
```

`ReadElement` and `WriteElement` need to check for abuse. `SemError(234)` is a new one; a suitable message for it has to be added to the `TOPSY.FRM` file.

```
  ReadElement
  =                           (. TABLE_types vartype;
                                 TABLE_entries entry; .)
     Variable<vartype, entry> (. if (entry.v.scalar) CGen->stackaddress(entry.v.offset);
                                 switch (vartype)
                                 { case TABLE_ints  : CGen->readintvalue(); break;
                                   case TABLE_bools : CGen->readboolvalue(); break;
                                   case TABLE_chars : CGen->readcharvalue(); break;
                                   case TABLE_none  : /* error already */   break;
                                          /* ------------- extended */
                                   default : SemError(211); break;
                                 } .) .

  WriteElement
  =                           (. TABLE_types exptype;
                                 char str[600];
                                 CGEN_labels startstring; .)
      String<str>             (. CGen->stackstring(str, startstring);
                                 CGen->writestring(startstring); .)
    | Expression<exptype>     (. switch (exptype)
                                 { case TABLE_ints  : CGen->writeintvalue(); break;
                                   case TABLE_bools : CGen->writeboolvalue(); break;
                                   case TABLE_chars : CGen->writecharvalue(); break;
                                   case TABLE_none  : /* error already */    break;
                                          /* ------------- extended */
                                   default : SemError(234); break;
                                 } .) .
```

Much of the manipulation of enumerated values as it relates to type casting is achieved in the extended `Factor parser`. There are some subtleties here. If a `Factor` starts with an identifier, this might be the familiar constant or variable; it might also be a casting function call like `COLORS(3)`. Effectively we have yet another potential LL(1) conflict, which can be resolved by driving the parser semantically, and using a `return` statement to break out in the cases where the identifier is not the name of an enumeration type. I guess you could have been forgiven for not spotting how to deal with this problem; it was a bit nasty (deliberately, to sort out the best from the mob!)

Type transfers such as `int(false)` or `bool(red)` or `char(45)` are more easily handled, as they are introduced by key words, and cause no LL(1) problems. This also applies in the case of the Modula-2 inspired val function. (I have shown both models for type transfers here; it was not necessary for candidates to use both of them.)

Notice that the type transfer functions probably do not need to generate code at all - a type cast or transfer is really nothing other than a way of fooling the type checking system imposed by the `compatible` check.

```
  Factor<TABLE_types &f>
  =                           (. int value;
                                 TABLE_entries entry;
                                          /* ------------- extended */
                                 TABLE_alfa name;
                                 bool found;
                                 TABLE_types ord; .)
       Designator<classset(TABLE_consts, TABLE_vars, TABLE_enums), entry, f>
                                          /* ------------- extended */
```

```
                              (. switch (entry.idclass)
                                 { case TABLE_vars :
                                     if (entry.v.scalar) CGen->stackvalue(entry.v.offset);
                                     else CGen->dereference();
                                     return;
                                   case TABLE_progs : return;
                                   case TABLE_enums : break;
                                   case TABLE_consts :
                                     CGen->stackconstant(entry.c.value); return;
                                 } .)
       ( "(" Expression<f> ")"   (. f = entry.type; .)
         |                       (. if (entry.idclass == TABLE_enums) SemError(220); .)
       )
     | Number<value>             (. CGen->stackconstant(value); f = TABLE_ints; .)
     | Char<value>               (. CGen->stackconstant(value); f = TABLE_chars; .)
     | "true"                    (. CGen->stackconstant(1); f = TABLE_bools .)
     | "false"                   (. CGen->stackconstant(0); f = TABLE_bools .)
                                                /* ------------ extended */
     | (   "int"                 (. ord = TABLE_ints; .)
         | "char"                (. ord = TABLE_chars; .)
         | "bool"                (. ord = TABLE_bools; .)
       ) "(" Expression<f> ")"   (. f = ord; .)
     | "val" "(" Ident<name>     (. Table->search(name, entry, found);
                                    if (!found && isalpha(name[0]))
                                    { SemError(202); // forced add to table
                                      strcpy(entry.name, name);
                                      entry.idclass = TABLE_enums;
                                      entry.e.maxValue = 0; entry.type = TABLE_none;
                                      Table->enter(entry);
                                    } .)
       "," Expression<f> ")"     (. f = entry.type; .)
     | "(" Expression<f> ")" .
```

However, if one is paranoid about safety, one might want to extend this further. For example, code like

```
            integer = int(bool(9) && bool(10));
```

is potentially dangerous, as one might not want not be allowed to regard 9 and 10 as values that can be cast to Boolean values, anded together and then cast to an integer value.

Nobody spotted this sort of thing; I did not expect that anybody would. Just for fun if we wnted to handle this we could provide more stack machine support in the form of yet another operation of the form

```
            RNG  Max
```

which will simply check the value on the top of the stack and abort the program if this is outside of the range `0 <= TOS <= Max.`

```
    case STKMC_rng:
      if (inbounds(cpu.sp))
        if (mem[cpu.sp] < 0 || mem[cpu.sp] > mem[cpu.pc]) ps = badval;
      cpu.pc++;
      break;
```

This code would be generated from within an extended form of the `Factor` parser by making a call onto a new code generating routine:

```
  void CGEN::rangecheck(int maxValue)
  // If maxValue != maxint, apply range check
  { if (maxValue != maxint && checking)
      { emit(int(STKMC_rng)); emit(maxValue); } }
```

where, it should be noted, we have introduced the facility for suppressing the paranoid check in conjunction with a Boolean flag `checking` that might be set by a pragma:

```
  PRAGMAS
    DebugOn     = "$D+" .          (. Report->debugging = true; .)
    RangeChecks = "$R+" .          (. CGen->checking = true; .)
  class CGEN {
    public:
      CGEN_labels undefined;     // for forward references
      bool checking;             // for strict range checking
```

The extended `Factor` production would then become:

```
  Factor<TABLE_types &f>
  =                            (. int value, maxValue;
                                  TABLE_entries entry;
                                             /* ------------ extended */
                                  TABLE_alfa name;
                                  bool found;
                                  TABLE_types ord; .)
       Designator<classset(TABLE_consts, TABLE_vars, TABLE_enums), entry, f>
                                             /* ------------ extended */
                               (. switch (entry.idclass)
                                  { case TABLE_vars :
                                      if (entry.v.scalar) CGen->stackvalue(entry.v.offset);
                                      else CGen->dereference();
                                      return;
                                    case TABLE_progs : return;
                                    case TABLE_enums : break;
                                    case TABLE_consts :
                                      CGen->stackconstant(entry.c.value); return;
                                  } .)
       ( "(" Expression<f> ")"  (. f = entry.type; CGen->rangecheck(entry.e.maxValue); .)
         |                      (. if (entry.idclass == TABLE_enums) SemError(220); .)
       )
     | Number<value>            (. CGen->stackconstant(value); f = TABLE_ints; .)
     | Char<value>              (. CGen->stackconstant(value); f = TABLE_chars; .)
     | "true"                   (. CGen->stackconstant(1); f = TABLE_bools .)
     | "false"                  (. CGen->stackconstant(0); f = TABLE_bools .)
                                               /* ------------ extended */
     | (   "int"                (. ord = TABLE_ints; maxValue = maxint; .)
         | "char"               (. ord = TABLE_chars; maxValue = 255; .)
         | "bool"               (. ord = TABLE_bools; maxValue = 1; .)
       ) "(" Expression<f> ")"  (. f = ord; CGen->rangecheck(maxValue); .)
     | "val" "(" Ident<name>    (. Table->search(name, entry, found);
                                  if (!found && isalpha(name[0]))
                                  { SemError(202); // forced add to table
                                    strcpy(entry.name, name);
                                    entry.idclass = TABLE_enums;
                                    entry.e.maxValue = 0; entry.type = TABLE_none;
                                    Table->enter(entry);
                                  } .)
       "," Expression<f> ")"    (. f = entry.type; CGen->rangecheck(entry.e.maxValue);  .)
     | "(" Expression<f> ")" .
```

Appendix - Topsy grammar - unattributed

```
COMPILER Topsy
/* Topsy++ level 1 grammar - no procedures, functions, parameters
   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.
```