RHODES UNIVERSITY

November Examinations - 2013

Computer Science 301 - Paper 1 - Solutions

Examiners:
Prof P.D. Terry
Prof P. Blignault

Time 4 hours
Marks 180
Pages 14 (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 an augmented version of "Section C" - a request to use Coco/R to generate C# code that could be incorporated within a complete program for analyzing the examination results of undergraduates, using a description of administrative actions to be taken (like exclusion) based on various criteria. Some 16 hours before the examination a complete grammar for a simple version of this system and other support files for building it were supplied to students, along with an appeal to study this in depth (summarized in "Section D"). During the examination, candidates were given machine executable versions of the Coco/R compiler generator, the files needed to build the basic systems, access to a computer, and machine readable copies of the questions.)

Section A: Conventional questions [90 marks]

A1. (a) What distinguishes a *context free grammar* from a *context sensitive grammar*? [2 marks]

In general, grammar productions take the form

$$\alpha \rightarrow \beta \qquad \text{with} \quad \alpha \in (N \cup T)^* N (N \cup T)^* \; ; \quad \beta \in (N \cup T)^*$$

where N is the set of non-terminals and T is the set of terminals. In context-free grammars $\alpha \in N$ *(the left side of each production must consist of a single non-terminal); for a grammar to be context-sensitive there must be at least one production for which* α *is comprised of two or more tokens. Note the importance of the phrase "at least one production". Sadly, few students knew, or seemed able to describe, this distinction.*

(b) The syntax of many programming languages is described by a context free grammar, and yet there are properties of most programming languages that are context sensitive. Mention one such property, and indicate briefly how this context sensitivity is handled in practical compilers. [4 marks]

There are plenty of examples to choose from - such as

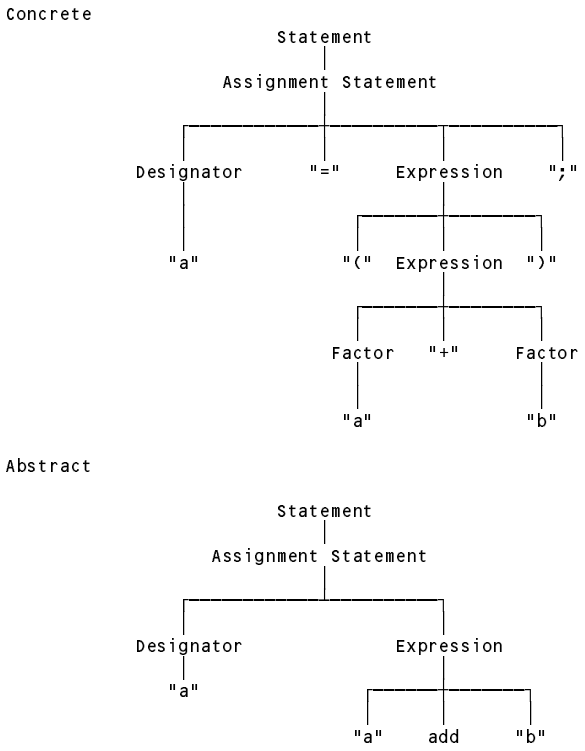
- *variables must be declared before they are used*
- *the number of formal and actual arguments for functions must agree*
- *expressions used to control if and while statements must be of Boolean type*
- *values assigned to variables must be of the appropriate type*
- *break statements may only be used in the context of a loop or switch statement*
- *a nice one suggested by this paper, question A7 - in a multiple assignment statement there must be as many "designators" as there are "expressions".*

Many (but not all) of these are usually handled by making use of a "symbol table" in which the various properties of the items denoted by identifiers are recorded, or by checking one count against another, or by parameterization of productions like the Statement production, needed for handling break statements correctly.

A2. What distinguishes a *concrete syntax tree* from an *abstract syntax tree*? Illustrate your answer by drawing both forms of tree corresponding to the simple Java statement [6 marks]

```
a = ( b + c ) ;
```

A concrete syntax tree incorporates all the tokens used in deriving the parse tree, while an abstract syntax tree retains only the information needed later to analyse the semantics completely:



A3. Describe briefly but clearly what you understand by the following two properties of variables as they are used in block structured imperative programming languages - firstly, their *scope* and, secondly, their *existence*. [4 marks]

The scope of an identifier in a block structured language is the area of code in which it can be recognized - typically the block in which it is declared (and any blocks nested within that block, subject to rules that might allow redeclaration on condition that the most recent declaration applies). Existence refers to the length of time at run time for which the variable has storage allocated to it - typically using a stack based runtime structure ensuring that such storage is allocated and deallocated as blocks are activated and deactivated.

A4. (a) What distinguishes a *native code compiler* from an *interpretive compiler*? [2 marks]

A native code compiler generates machine level object code, typically for the same machine that is executing the compiler itself. An interpretive compiler generates intermediate level code, typically for a virtual machine whose operation can be emulated by a suitable interpreter for such code.

(b) Suggest (with some explanation) one property of native code compilation that is claimed to offer an advantage over interpretive compilation, and also one property of interpretive compilation that is claimed to offer an advantage over native code compilation. [2 marks]

Native code compilers should produce object code that can be executed at the full speed of the host machine, typically one or two orders of magnitude faster than an interpreter can execute code for a virtual machine. Interpretive compilers on the other hand are much more easily developed, and can be made highly portable (all that is needed is a suitable interpreter for the virtual code).

(c) What do you understand by the technique known by the acronym JIT, and what systems known to you incorporate this technique? [4 marks]

JIT stands for "just in time" and refers to the technique frequently used by Java and .NET systems of completing the translation of code for a virtual machine or interpreter into native machine code only when it is needed.

A5. *(Cocol and recursive descent parsing)* The following familiar Cocol grammar describes a set of EBNF productions (of the form found in the PRODUCTIONS section of the grammar itself).

```
COMPILER EBNF $CN
/* Describe a set of EBNF productions
   P.D. Terry, Rhodes University, 2013 */

CHARACTERS
  eol         = CHR(10) .
  space       = CHR(32) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  lowline     = "_" .
  printable   = ANY - control .
  nonSpace    = printable - space .
  noquote1    = ANY - "'" - control .
  noquote2    = ANY - '"' - control .

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

IGNORE control

PRODUCTIONS
  EBNF        = { Production } EOF .
  Production  = nonterminal "=" Expression  "." .
  Expression  = Term { "|" Term } .
  Term        = Factor { Factor } .
  Factor      =   nonterminal | terminal | "(" Expression ")"
                | "[" Expression "]" | "{" Expression "}" .
END EBNF.
```

(a) In the original notation known as BNF, productions took a form exemplified by

```
<nonterminal> ::= ε  |  terminal1  | ( <another nonterminal> | terminal2 ) + <something>
```

The notation allowed the use of an explicit ε. [] brackets and {} braces were not used (although () parentheses were allowed). Non-terminals and terminals were distinguished by the presence or absence of < > angle brackets, and a production was terminated at the end of a line.

Describe a set of BNF productions using Cocol (simply modify the grammar above). Use `"eps"` to represent ε. [15 marks]

The transformation of the PRODUCTIONS section is straightforward. We have to redefine the form that terminals and non-terminals take in the TOKENS section. This is the trickiest part to get right, but something like this had been discussed in tutorials and tests, and I was hoping for something on the lines of the following:

```
COMPILER BNF $CN
/* Describe a set of BNF productions
  P.D. Terry, Rhodes University, 2013 */

CHARACTERS
  eol         = CHR(10) .
  space       = CHR(32) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  lowline     = "_" .
  printable   = ANY - control .
  nonSpace    = printable - space .
  startTerm   = printable - "<" .

TOKENS
  nonterminal = "<" letter { letter | lowline | digit | space } ">" .
  terminal    = startTerm { nonSpace } | "<"  | '"("' | '")"' .
  EOL         = eol .

IGNORE control - eol
```

```
PRODUCTIONS
  BNF         = { Production } EOF .
  Production  = nonterminal "::=" Expression EOL .
  Expression  = [ "ε" "|" ] Term { "|" Term } .
  Term        = Factor { Factor } .
  Factor      = nonterminal | terminal | "(" Expression ")" .
END BNF.
```

This question must have been badly understood or misread by some students. They were asked to describe BNF notation using Cocol, not to write a description of EBNF using BNF - and certainly were not being asked to write a grammar that had eliminated all the meta-brackets.

Note where the null option ε has been introduced! It can be introduced as an alternative in Factor, but this then permits multiple εs to appear in an Expression. (No candidates saw this subtlety, nor did I expect them to do so).

Many candidates were confused by the example production. In retrospect it would have been better had I used examples like these

```
<Full Name>     ::= <Title> Terry <Qualification>
<Title>         ::=  ε  | ( Mr  | Dr | Prof )
<Qualification> ::= PhD "(" Cantab ")"
```

- the point being that, effectively, anything not in < > brackets is a terminal, which leads to a small problem in representing the meta-symbols (| and). All the more reason why Niklaus Wirth's EBNF is that much easier to work with!

(b) Assume that you have available a suitable scanner method called `getSym` that can recognize the terminals of BNF and classify them appropriately as members of the following enumeration

```
EOFSym, noSym, EOLSym, termSym, nontermSym, definedBySym,
epsilonSym, barSym, lParenSym, rParenSym
```

Develop a hand-crafted recursive descent parser for recognizing a set of BNF productions based on your description in (a). *(Your parser can take drastic action if an error is detected. Simply call methods like* `accept` *and* `abort` *to produce appropriate error messages and then terminate parsing. You are not required to write any code to implement the* `getSym`, `accept` *or* `abort` *methods.)* [15 marks]

What was expected was code on the following lines (and provided students had the grammar right, this followed easily):

```
static IntSet FirstFactor = new IntSet(lParenSym, termSym, nontermSym});

static void BNF() {
  while (sym == nontermSym) {
    Production();
  }
  accept(EOFSym, "EOF expected");
}

static void Production() {
  getSym();
  accept(definedBySym, "::= expected");
  Expression();
  accept(EOLSym, "productions must be terminated by end-of-line");
}

static void Expression() {
  if (sym == epsilonSym) {
    getSym();
    accept(barSym, "| expected");
  }
  Term();
  while (sym == barSym) {
    getSym(); Term();
  }
}
```

```
static void Term() {
  Factor();
  while (FirstFactor.contains(sym)) Factor();
}

static void Factor () {
  switch (sym) {
    case nontermSym :
    case termSym :
      getSym();
      break;
    case lParenSym :
      getSym(); Expression(); accept(rParenSym, ") expected");
      break;
    default:
      abort("invalid start to Factor");
      break;
  }
}
```

A6. *(Grammars)* By now you should be familiar with RPN or "Reverse Polish Notation" as a notation that can describe expressions without the need for parentheses. The notation eliminates parentheses by using "postfix" operators after the operands. To evaluate such expressions one uses a stack architecture, such as formed the basis of the PVM machine studied in the course. Examples of RPN expressions are:

```
3 4 +                    - equivalent to    3 + 4
3 4 5 + *                - equivalent to    3 * (4 + 5)
```

In many cases an operator is taken to be "binary" - applied to the two preceding operands - but the notation is sometimes extended to incorporate "unary" operators - applied to one preceding operand:

```
4 sqrt                   - equivalent to    sqrt(4)
5 -                      - equivalent to    -5
```

The following represent two attempts to write a grammar for an RPN expression:

```
(G1)    RPN      =     RPN RPN binOp
                     | RPN unaryOp
                     | number .
        binOp    =   "+" | "-" | "*" | "/" .
        unaryOp  =   "-" | "sqrt" .

(G2)    RPN      =   number REST .
        REST     =   [ number REST binOp REST | unaryOp ].
        binOp    =   "+" | "-" | "*" | "/" .
        unaryOp  =   "-" | "sqrt" .
```

(a) Give a precise definition of what you understand by the concept of an *ambiguous grammar*. [2 marks]

(b) Give a precise definition of what you understand by the concept of *equivalent grammars*. [2 marks]

An ambiguous grammar is one for which at least one sentence can be derived in more than one way, that is, for which the parse tree is not unique.

Grammars are equivalent if they derive exactly the same set of sentences (not necessarily using the same set of sentential forms or parse trees).

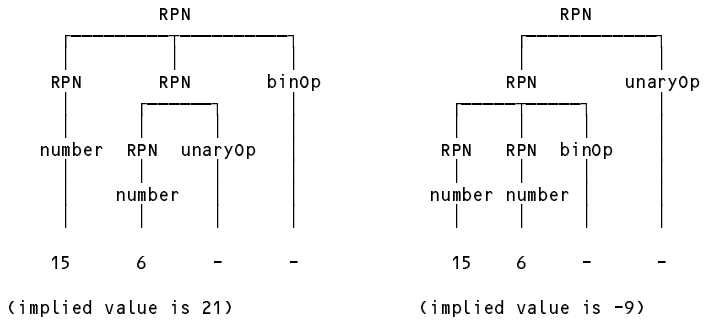
(b) Using the expression

```
15  6  -  -
```

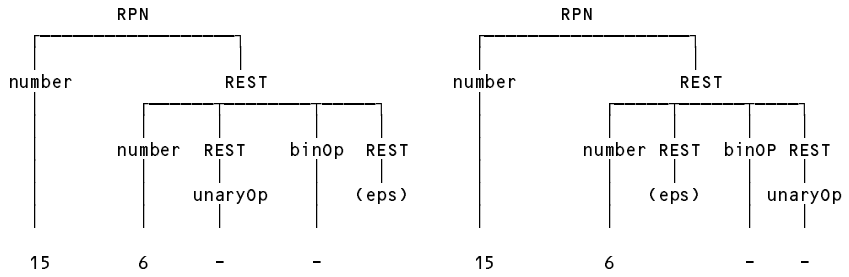
as an example, and by drawing appropriate parse trees, demonstrate that both of the grammars

above are ambiguous. [6 marks]

Nearly all candidates could do this one correctly! Using grammar G1:



Using grammar G2:



(d) Analyse each of these grammars to check whether they conform to the LL(1) conditions, explaining quite clearly (if they do not!) where the rules are broken. [6 marks]

G1 is left recursive and this cannot be an LL(1) grammar. There are two alternatives for the right side of the production for RPN that both start with RPN, so Rule 1 is broken

*For G2, REST is nullable. First(REST) is { number, sqrt, - } while Follow(REST) is { +, -, / , * } so Rule 2 is broken.*

A7. *(Code generation)* A BYTE (Bright Young Terribly Eager) student has been nibbling away at writing extensions to her first Parva compiler, while learning the Python language at the same time. She has been impressed by a Python feature that allows one to write multiple assignments into a single statement, as exemplified by

```
A, B = X + Y, 3 * List[6];
A, B = B, A;                // exchange A and B
A, B = X + Y, 3, 5;         // incorrect
```

which she correctly realises can be described by the context-free production

```
Assignment = Designator { "," Designator } "=" Expression { "," Expression } ";"
```

The Parva attributed grammar that she has been given deals with single, simple integer assignments only:

```
Assignment                      (. DesType des; .)
=  Designator<out des>          (. if (des.entry.kind != Entry.Var)
                                      SemError("invalid assignment"); .)
   "="
   Expression                   (. CodeGen.assign(); .)
   ";" .
```

where the `CodeGen.assign()` method generates an opcode that is matched in the interpreter by the following `case` arm:

```
case PVM.sto:
  //   store value at top of stack on address at second on stack
  mem[mem[cpu.sp + 1]] = mem[cpu.sp];

  //   bump stack pointer
  cpu.sp = cpu.sp + 2;
  break;
```

Suggest, in as much detail as time will allow, how the `Assignment` production and the interpreter would need to be changed to support this language extension. Give the Cocol and C#/Java code that would be needed, not a simple hand-waving argument like "add some more opcodes". [20 marks]

To ensure that there are as many expressions as designators it is easiest to count both, and then make a simple check immediately before the terminating semicolon is identified. This count can be used as an argument to a new `PVM.sto` *opcode:*

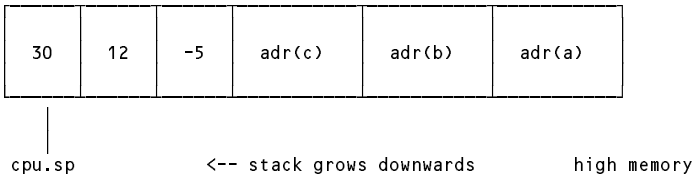
```
          Assignment                      (. DesType des;
                                             int desCount = 1, expCount = 1; .)
***       =  Designator<out des>          (. if (des.entry.kind != Entry.Var)
                                               SemError("invalid assignment"); .)
***          { "," Designator<out des>    (. if (des.entry.kind != Entry.Var)
***                                            SemError("invalid assignment");
***                                         desCount++; .)
***          }
             "="
             Expression
***          { "," Expression             (. expCount++; .)
***          }                            (. if (expCount != desCount)
***                                            SemError("left and right counts disagree");
***                                         CodeGen.Assign(desCount); .)
             ";" .
```

where the action of the new `PVM.sto` *opcode is to loop through the appropriate number of assignments and then to pop 2n values off the operand stack:*

```
case PVM.sto:
  //   store n values at top of stack on address at second on stack
  int n = Next();
  for (int i = n - 1; i >= 0; i--)
    mem[mem[cpu.sp + n + i] = mem[cpu.sp + i];

  //   bump stack pointer
  cpu.sp = cpu.sp + 2 * n;
  break;
```

Perhaps it might help to illustrate the state of the stack immediately before the STO 3 opcode is executed, for the simple case of `a, b, c = -5, 5 + 7, 30 ;`



There are other ingenious ways of solving this problem. One candidate stacked up the addresses, then reversed that stack segment, which allowed for the use of the usual STO opcode to be applied 3 times as the three expressions were calculated. Still others found ways to use STO M instructions in which M decreased as each expression was calculated.

All of which made me start to think "language lawyer" again. Should one be allowed to write statements like

```
a, a, a   =   3, a + 4, a - 7 ;
```

for example, how obvious would their semantics be, and how annoying or restrictive would this potentially useful statement be if some non-intuitive semantics were demanded and imposed. This year's creative candidates have given me some good ideas for next year's compiler practicals and tutorials. Here is another one to chew over. Should

```
des1, des2, des3  =   exp1, exp2, exp3;
```

be semantically equivalent to the sequence

```
des1 = exp1;
des2 = exp2;
des3 = exp3;
```

or to the sequence

```
des3 = exp3;
des2 = exp2;
des1 = exp1;
```

or does it (and should it) make no difference? If it appears to make a difference, is there any way of preventing it from doing so?

Which (if either) of these alternatives corresponds to the solution suggested above, or does this correspond to yet another semantic interpretation?

Kinky language features are not always as simple or useful as they might at first appear. The other well known statement which at first sight seems very simple is the "for" loop, which can keep language lawyers amused for hours. In some years I have used it as an example, but this year did not do so.

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 EXAMJ.ZIP (Java) or EXAMC.ZIP (C#) modify any files that you choose, and then copy all the files back onto an exam folder on the network.

This section is based on a real system that I am trying to develop. As such, this section represents possibly not only the most realistic, but also the most open-ended and difficult of all the problems ever set in this series. I was amazed and delighted by the depth of maturity of many answers - especially as the class had reached that maturity in only a few weeks.

QUESTION B8 [3 marks]

We have used the operators `and, or, not, =` and `/=` to denote operations which in C# or Java are denoted by `&&, ||, !, ==` and `!=`. This is reasonable; the suggestions might be more in line with what Deans readily understand. How would the system be changed to allow *either* version of each of these operators to be used in the criteria list (for the benefit of the Dean of Commerce, who just happens to be a programmer).

All that is needed is to change the definitions of the various operators:

```
EqualOp<out int op>                 (. op = CodeGen.nop; .)
=    ( "=" | "==" )                 (. op = CodeGen.ceq; .)
   | ( "/=" | "!=" )                (. op = CodeGen.cne; .)
.

NotOp<out int op>                   (. op = CodeGen.not; .)
=  "not" | "!"
.

AndOp<out int op>                   (. op = CodeGen.and; .)
=  "and" | "&&"
.

OrOp<out int op>                    (. op = CodeGen.or; .)
=  "or"  | "||"
.
```

QUESTION B9 **[2 marks]**

Other than simply calling it a "compiler" or "translator", how would you classify the system that is built for translating specifications to code (for example: pretty-printer, assembler, decompiler, native code compiler, interpretive compiler, *etc* …)?

It's probably best described as a high-level compiler.

QUESTION B10 **[2 marks]**

It was suggested in the preliminary discussion that you could write your system in Java, although the rest of the larger system into which your generated method would eventually be injected was originally written in C#. Presumably a Java implementation could be arranged to generate C# code. What is the term used to describe translators that generate code for a system other than the one in which they are themselves implemented?

A cross-compiler.

QUESTION B11 **[10 marks]**

The `Table` class supplied as a possibility in the exam kit is weak in several respects. In particular it does not check that any identifier can be added to the table only once, or that the actions have tag numbers that are distinct. Suggest how it could be improved in this respect (give code modifications, not vague descriptions).

For redeclared identifiers we need a simple search of the symbol table for an existing entry (which should fail, and return the sentinel entry. For a non-unique action number we can make use of the `allActions` *set which was added as a "plum" in the solution released the previous day.*

```
     public static void Insert(Entry entry) {
     // Adds entry to symbol table
***    Entry check = Find(entry.name);
***    if (check != sentinelEntry) {
***      Console.WriteLine(entry.name + " redeclared");
***      System.Environment.Exit(12);
***    }
***    if (entry.kind == Kinds.action && allActions.Contains(entry.actionNumber)) {
***      Console.WriteLine(entry.name + " - action number " + entry.actionNumber + " previously assigned");
***      System.Environment.Exit(12);
***    }
       symbolTable.Add(entry);

       allActions.Incl(entry.actionNumber);
       if (entry.actionNumber > highestActionNumber)
         highestActionNumber = entry.actionNumber;
     } // Table.Insert

     public static Entry Find(string name) {
     // Searches table for an entry matching name.  If found then return that
     // entry; otherwise return the sentinel entry (marked as not declared).
       symbolTable[0].name = name;
       int look = symbolTable.Count - 1;
       while (!name.Equals(symbolTable[look].name)) look--;
       return symbolTable[look];
     } // Table.Find
```

QUESTION B12 **[3 marks]**

It might be thought helpful to allow a Dean to use synonyms (words with the same meaning) to define criteria. Suppose, for example, that we wanted to be able to use any of the words `Lowest, Lowest_MarkYear` or `Worst` to refer to a student's lowest mark for the year. How could this feature be added to your system (give code if possible)?

All that is needed is to add lines to the `Table.txt` *file. There is no difficulty with having extra entries that have the same field names associated with them. provided that their keys are distinct.*

```
Field        Lowest                    int       s.lowestMark
Field        Lowest_MarkYear           int       s.lowestMark
Field        Worst                     int       s.lowestMark
```

QUESTION B13 **[10 marks]**

The system as provided implies that a set of criteria applies to a single Faculty. There is no check made that a student being analysed is, in fact, registered in that Faculty. Modify the code to detect such errors, and add suitable warning messages to the output log file if they occur.

We simply generate code that compares the faculty name with the student's faculty at run time. If a mismatch occurs, flag an error and add the student's summary to the logfile, and simply ignore the rest of the `BuildSet` *code (note the "return" statement that is generated).*

```
      Converter                   (. Entry entry; .)
***   = Faculty<out entry>        (. CodeGen.Append("if (!s.faculty.Equals(" + entry.codedName + ") ");
***                                  CodeGen.AppendLBrace;
***                                  CodeGen.Append("logFile.WriteLine(s.summary");
***                                  CodeGen.Append(" + \" is not in \" + entry.name);");
***                                  CodeGen.AppendNewLine();
***                                  CodeGen.Append("return;");
***                                  CodeGen.AppendRBrace(); .)
      // rest as before ....
```

It might be neater to code this as a call to a method in the CodeGen class:

```
      Converter                   (. Entry entry; .)
***   = Faculty<out entry>        (. CodeGen.CheckFaculty(entry); .)
      // rest as before ....

      public static void CheckFaculty(Entry faculty) {
      // Generate code to check that the current student is from the stipulated faculty
***     Append("if (!s.faculty.Equals(" + faculty.codedName + ") ");
***     AppendLBrace;
***     Append("logFile.WriteLine(s.summary + \" is not in \" + faculty.name);");
***     AppendNewLine();
***     Append("return;");
***     AppendRBrace(); .)
      } // CodeGen.CheckFaculty
```

QUESTION B14 **[15 marks]**

(Use your imagination) Suppose a Dean writes a set of criteria like the following

```
FirstYear:
  Exclude :       Passes < 2;
                  Lowest_Mark < 15.
  Congratulate :  Passes = Courses.
```

In this case a student who had taken only one subject, and passed it at 50% would be marked for both congratulation and exclusion, which is kind of silly. In this case the silliness is easy to spot, but in general it might be far more difficult to do so. Can you think of any mechanism that the Ancient Skilled Programmer might recommend that the Deans might use so as to allow the system to detect and report nonsensical combinations of actions, perhaps by generating a suitable entry to the log file. (Give appropriate code where possible.)

Given the entry in the table.txt file, intended as another plum to tempt sharp-eyed readers of the solution released during the afternoon of the day before:

```
Action        INCONSISTENCY               bool      0
```

I had hoped that someone might suggest that the Dean be advised to write criteria like

```
FirstYear:
  Exclude :       Passes < 2;
                  Lowest_Mark < 15.
  Congratulate :  Passes = Courses.
  Inconsistency:  Exclude and Congratulate.
```

and then, after analyzing a student, complain if the number 0 (INCONSISTENCY) appeared in the action set. This could have been achieved by modifying the CodeGen.Epilogue() method to read:

```
            StartNewLine();
            codeFile.Write("s.actionString = BuildActionString(s.actions);");
      ***   StartNewLine();
      ***   codeFile.Write("if (s.actions.Contains(INCONSISTENCY)) {");
      ***   StartNewLine();
      ***   codeFile.Write("  logFile.WriteLine(s.summary + \" - INCONSISTENT ACTIONS\");");
      ***   StartNewLine();
      ***   codeFile.Write("}");
            StartNewLine();
            AppendRBrace();

            codeFile.WriteLine(" // ActionSetBuilder.BuildSet");
            codeFile.WriteLine();
```

However, this leaves a lot to be desired. As many people noticed - spurred on by question 17, which with hindsight should have preceded question 14 - this would simply not work if criteria were specified in a bad order:

```
          FirstYear:
            Inconsistency:  Exclude and Congratulate.
            Exclude :       Passes < 2;
                            Lowest_Mark < 15.
            Congratulate :  Passes = Courses.
```

Inconsistencies are not really actions, and probably should not be put into the symbol table in the same way as other actions. Furthermore, they really apply to all students in all faculties in all years. A much better approach would be to introduce Inconsistency as a key word, and write a grammar like the following (some previous checks omitted to save space):

```
      Converter                   (. Entry entry; .)
      = Faculty<out entry>
        {   "FirstYear"  ":" CriteriaList<1>
          | "SecondYear" ":" CriteriaList<2>
          | "ThirdYear"  ":" CriteriaList<3>
          | "FourthYear" ":" CriteriaList<4>
          | "AnyYear"    ":" CriteriaList<0>
        SYNC
        }
***     Inconsistency .

***   Inconsistency
***   = { "Inconsistency" ":"     (. CodeGen.StartIf();
***                                  CodeGen.AppendLParen(); .)
***       Condition               (. CodeGen.AppendRParen(); .)
***       { WEAK ";"              (. CodeGen.AppendOperator(CodeGen.or);
***                                  CodeGen.AppendLParen(); .)
***         Condition             (. CodeGen.AppendRParen(); .)
***       } SYNC "."              (. CodeGen.AppendRParen();
***                                  CodeGen.StartNewLine();
***                                  CodeGen.Append("  s.actions.Incl(INCONSISTENCY);");
***                                  CodeGen.StartNewLine(); .)
***     } .
```

Having said that, there were come innovative and intriguing suggestions mad by candidates. For example, there was a suggestion that actions fell into three categories - punishments, reward, and neutral - and that the action numbers should take these into effect. Simple tests on the intersections of a student's action set with each of the two sets corresponding to rewards and to punishments should produce two sets whose further intersection should be empty. Another suggestion is to amend the table so that each action line defines two sets - the singleton set for that action and then another set, possibly empty, of actions which would be inconsistent with that one - so for example we might have

```
# Kind (14)   Name (28)                Type (9)  Tag
#
Action        Exclude                  bool      10   { 11, 12, 14 }
Action        Congratulate             bool      11   { 10, 12, 13 }
Action        Probation                bool      12
Action        Fails_Rule_G7            bool      13   ( 11, 12, 14
Action        Gets_Degree              bool      14
```

Something on these lines is probably a very good way to handle the problem, as there is no real need to stipulate fixed sets that need to be computed each time the system is run.

QUESTION B15 **[10 marks]**

Is it really necessary to use semicolons between alternative criteria and to terminate a set of alternatives with a period (full stop)? Deans might regard these as confusing punctuation and prefer something simpler such as

```
FirstYear:
  Exclude :       Passes < 2
                  Lowest_Mark < 15
  Congratulate :  Passes = Courses
```

If punctuation is unnecessary, how would the attributed grammar be changed to accommodate specifications like these.

The semicolon is necessary but not the period. It is very easy to drop the punctuation, as shown below.

```
        CriteriaList<int year>        (. int action; .)
        =                             (. CodeGen.StartCriteriaList(year); .)
          { Action<out action> ":"    (. CodeGen.StartIf();
                                          actionsUsed.Incl(action); .)
            Condition                 (. CodeGen.EndIf(action); .)
***         {                         (. CodeGen.AppendNewLine();
                                          CodeGen.StartIf(); .)
              Condition               (. CodeGen.EndIf(action); .)
***         }                         (. CodeGen.AppendNewLine(); .)
          }                           (. CodeGen.EndCriteriaList(); .) .
```

Since a `Condition` *can (although probably rarely does) start with a + sign, dropping the semicolon would lead to LL(1) violations. However, retaining the semicolon and discarding the period leads to no such error. (Why not?) Few students, if any, spotted this or offered an LL(1) based argument. Rather more interestingly, several suggested that if the semicolons were to be discarded, one should introduce EOL as a delimiter, on the lines of*

```
        CriteriaList<int year>        (. int action; .)
        =                             (. CodeGen.StartCriteriaList(year); .)
          { Action<out action> ":"    (. CodeGen.StartIf();
                                          actionsUsed.Incl(action); .)
            Condition                 (. CodeGen.EndIf(action); .)
***         { WEAK EOL                (. CodeGen.AppendNewLine();
                                          CodeGen.StartIf(); .)
              Condition               (. CodeGen.EndIf(action); .)
***         } SYNC "."                (. CodeGen.AppendNewLine(); .)
          }                           (. CodeGen.EndCriteriaList(); .) .
```

From a compiler writer's perspective, discuss whether there is any merit in insisting that punctuation be used.

Punctuation marks often give useful synchronization (error recovery) points, as most candidates pointed out.

QUESTION B16 **[15 marks]**

In practice, Faculties administer several degrees - for example at Rhodes the Commerce Faculty administers BCom, BEcon and BBS. The criteria used by Deans often apply to some degrees, but not all.

Suppose we wished to extend the system to allow input exemplified by

```
Humanities
Degrees: BA, BSS:
  FirstYear:
    Exclude: Passes < 2.
  SecondYear:
    Exclude: Full_Creds_So_Far < 3.
Degrees: BFA, BJrn, BMus:
  FirstYear:
    Exclude: Passes < 3.
AllDegrees:
  SecondYear:
    Exclude: Full_Creds_SoFar < 6.
```

That is, where some degrees have one common set of criteria, while some other degrees have a different common set. In terms of the unattributed grammar this might be achieved by altering the `Converter` production to read:

```
Converter = Faculty
            ( ( "AllDegrees" ":" | "Degrees" ":" Degree { "," Degree } ":" )
              {   "FirstYear"  ":" CriteriaList
                | "SecondYear" ":" CriteriaList
                | "ThirdYear"  ":" CriteriaList
                | "FourthYear" ":" CriteriaList
                | "AnyYear"    ":" CriteriaList
              }
            } .
```

Complete the remaining changes that would be needed to allow this extension.

This can be done as below. The question was designed simply to test whether the candidates could write simple actions. Unfortunately the question paper had an error, and the parentheses and braces appeared in the wrong places, with the result that a few students who actually tried to code this for real ended up with spurious errors. One way of coding the necessary additions would be

```
      Converter                   (. Entry entry; .)
      = Faculty<out entry>        (. CodeGen.CheckFaculty(entry); .)
***     { (   "AllDegrees" ":"
***         | "Degrees" ":"       (. CodeGen.Append("if ("); .)
***           Degree<out entry>   (. CodeGen.Append(entry.codedName + ".Equals(s.degree)"); .)
***           { ","               (. CodeGen.Append(" || "); .)
***             Degree<out entry> (. CodeGen.Append(entry.codedName + ".Equals(s.degree)"); .)
***           } ":"               (. CodeGen.Append(") "); .)
***       )                       (. CodeGen.AppendLBrace(); .)
          {   "FirstYear"  ":" CriteriaList<1>
            | "SecondYear" ":" CriteriaList<2>
            | "ThirdYear"  ":" CriteriaList<3>
            | "FourthYear" ":" CriteriaList<4>
            | "AnyYear"    ":" CriteriaList<0>
***       }                       (. CodeGen.AppendRBrace(); CodeGen.StartNewLine(); .)
        }                         (. CodeGen.AppendRBrace(); .)  .
```

Incidentally, even though several students did it this way, nobody commented on the fact that alternative criteria in the `CriteriaList` *production could also have been handled by a sequence of "ors" rather than a sequence of IfStatements. I wonder - what might have been the approach taken by students on the previous day?*

QUESTION B17 **[20 marks]**

The system as it stands allows Deans to write criteria like

```
SecondYear:
  Weak    : Passes < 3;
  Exclude : Weak or Average < 30.
```

where one criterion might depend upon another one. It also attempts to warn a Dean who might have written those particular criteria in the reverse order:

```
SecondYear:
  Exclude : Weak or Average < 30;
  Weak    : Passes < 3.
```

(To understand why such a check might be useful, consider a student who has taken five subjects and failed them all at 45%.)

Here is another, more complicated example:

```
FirstYear :
  Weak : Average < 40.
SecondYear:
  Exclude : Weak or Average < 30.
  Weak : Passes < 3.
```

(a) What well-known feature of many computer languages is being called into play here? [2 Marks]

I was hoping that students would see this as "declare and define before other use". Some answered "context sensitivity" which is correct, if a little too broad, while still others offered "scope" which is certainly part of the issue at hand.

(b) In a study of translators a distinction is drawn between *static semantics* and *dynamic semantics*. Does the strategy suggested in the grammar supplied to you qualify to be termed a static semantic check or a dynamic semantic check? Justify your answer. [4 Marks]

The translator is attempting to check as it compiles a set of criteria, whether these obey the same sort of semantic constraints demanded by other languages where variables must be declared before use. Here, rather unusually, the "symbol table" is not constructed as variables are declared, but instead a simple "scope" in the form of a set is constructed as the predeclared action names are encountered, effectively marking them as "now properly declared". The other predeclared names in the symbol table are, effectively, constants, and for these the problem does not arise.

Many candidates felt that it was a "dynamic semantic" check. As we are about to see, a proper attack on the problem requires such checking, but not for the reasons they thought. Sure, any errors will show up only when a program is executed, but in the case of a compiler, the errors it finds at compile time are "syntactic" and "static semantic". Only very sophisticated compilers can predict dynamic semantic violations accurately.

(c) Does the suggested strategy actually work? Explain your reasoning whether you believe it does or does not (don't simply guess!) If not, can you suggest how it might be improved, and discuss whether your improvements would always solve the problem in situations like these. If not, why not? (We are looking for insight here!) [14 Marks]

Unfortunately it does not always work. The strategy has only been able to create a set of all the actions that have ever been declared in any of the criteria, checking each time any action is referred to in an expression, that this action is in that set. What is needed is to recognize that the "scope" is, more realistically, controlled by the year of study (a student can be in only one year of study). If the `Parser.actionsDefined` *set were to be re-initialized each time a new set of constraints was encountered, problems such as suggested by the example*

```
FirstYear :
  Weak : Average < 40.
SecondYear:
  Exclude : Weak or Average < 30.
  Weak : Passes < 3;
```

would be solved. Alas, a compiler writer's life is not that easy. This is an easy change to make, but it does not help that much. Our system will also permit specifications like

```
FirstYear :
  Weak : Average < 40.
SecondYear:
  Exclude : Weak or Average < 30.
  Weak : Passes < 3.
FirstYear :
  Probation : Weak or Passes = 1.
```

which are surely acceptable. Pursuing this train of thought would seem to imply that the whole concept of "block structured scoping" needs to be handled in some way.

Not a prospect that would fill students with joy in a 4 hour exercise. However, there were many submissions who could see that some sort of scoping system was needed, and I was delighted with that.

A draconian approach would be to redefine the specification language with the criteria limited by a far narrower grammar, on the lines of

```
Converter = Faculty
            [ "FirstYear"  ":" CriteriaList ]
            [ "SecondYear" ":" CriteriaList ]
            [ "ThirdYear"  ":" CriteriaList ]
            [ "FourthYear" ":" CriteriaList ] .
```

Another attempt at improving the comes from the insight that we can handle the scope issues and the "define before use" semantic requirements at run-time. For each student we construct a set of "now defined" criteria in the `defined` *field of the* `Student` *class, which candidates might not have noticed already contained fields:*

```
***     public IntSet  defined = new IntSet();  // A set of integers corresponding to defined actions
***     public IntSet  actions = new IntSet();  // A set of integers corresponding to recommended actions
***     public string  actionString;            // Summary of recommendation actions as a concatenated string
```

As each action is defined it is added into the `s.defined` *set; if the criterion is met, it is also added to the* `s.actions` *set:*

```
        CriteriaList<int year>       (. int action = 0; .)
        =                            (. CodeGen.StartCriteriaList(year); .)
          { Action<out action> ":"   (. CodeGen.StartIf(); .)
            Condition                (. CodeGen.EndIf(action); .)
            { WEAK ";"               (. CodeGen.StartNewLine();
                                        CodeGen.StartIf(); .)
              Condition              (. CodeGen.EndIf(action); .)
            } SYNC "."               (. CodeGen.StartNewLine(); .)
***                                  (. CodeGen.Append("s.defined.Incl(" + action + ");");
          }                          (. CodeGen.EndCriteriaList();
                                        CodeGen.StartNewLine(); .) .
```

But this is only part of the story. To check at run-time that criteria definitions do not refer to as-yet undefined other actions requires that, as one evaluates a condition, one tracks the actions used in that condition. Globally within the `Parser class` *we define (how do we get away with a global field?):*

```
        static IntSet actionsUsed;
```

and add to this set when the `Designator` *parser encounters a name denoting an action:*

```
        Designator<out int type>     (. Entry entry; .)
        = Identifier<out entry>      (. if (entry.kind == Kinds.action) {
                                          CodeGen.Append("s.actions.Contains(" + entry.actionNumber + ")");
***                                       actionsUsed.Incl(entry.actionNumber);
                                     }
                                     else CodeGen.Append(entry.codedName);
                                     type = entry.type; .) .
```

The `CriteriaList` *production initialises this set as each action is encountered and then interrogates it once all the conditions have been evaluated:*

```
        CriteriaList<int year>       (. int action = 0; .)
        =                            (. CodeGen.StartCriteriaList(year); .)
***       { Action<out action> ":"   (. actionsUsed = new IntSet();
                                        CodeGen.StartIf(); .)
            Condition                (. CodeGen.EndIf(action); .)
            { WEAK ";"               (. CodeGen.StartNewLine();
                                        CodeGen.StartIf(); .)
              Condition              (. CodeGen.EndIf(action); .)
            } SYNC "."               (. CodeGen.StartNewLine(); .)
***                                  (. if (action != 0 && !actionsUsed.IsEmpty()) {
***                                       CodeGen.CheckDefinedActions(NewSet(actionsUsed));
***                                       CodeGen.StartNewLine();
***                                  }
***                                     CodeGen.Append("s.defined.Incl(" + action + ");");
                                        CodeGen.StartNewLine(); .)
          }                          (. CodeGen.EndCriteriaList();
                                        CodeGen.StartNewLine(); .) .
```

A little helper method is added to the parser grammar:

```
static string NewSet(IntSet s) {
// Constructs a string representing a new set constructor for the elements in s
  string str = s.ToString();
  return "new IntSet(" + str.Substring(1, str.Length - 2) + ")";
} // Parser.NewSet
```

When the system detects that one action depends on other actions, the `CheckDefinedActions` *method adds calls to a new method in the* `ActionSetBuilder` *class:*

```
  public static void CheckDefinedActions(string actionsUsed) {
    Append("CheckDefined(logFile, s.summary, s.defined, " + actionsUsed + ");");
  } // CheckDefinedActions
```

Source for this method is generated by additions to the CodeGen.Epilogue() method:

```
// Construct a method to log undefined actions used in conditions

codeFile.WriteLine();
codeFile.WriteLine("  public static void CheckDefined(OutFile logFile, string message, IntSet defined, IntSet
codeFile.WriteLine("  // Log any actions that were used but never previously defined, with a suitable message
codeFile.WriteLine();
codeFile.WriteLine("    if (!used.IsEmpty() && !used.Difference(defined).IsEmpty()) {");
codeFile.WriteLine("      logFile.WriteLine();");
codeFile.WriteLine("      logFile.WriteLine(message + \" - undefined criteria \"");
codeFile.WriteLine("                        + BuildActionString(used.Difference(defined)));");
codeFile.WriteLine("    }");
codeFile.WriteLine();
codeFile.WriteLine("  } // ActionSetBuilder.CheckDefined");
```

It should be emphasized that we have sacrificed the advantages of a compile-time check for the nuisance of a run-time disaster. In a practical application of this system one would probably have to make several passes over the student data, isolating the badly specified criteria and then changing the specification accordingly.

And finally, it should be added that although the Converter.atg grammar and the CodeGen and Student classes supplied in the examination had cunningly incorporated all the "plums" needed to build this solution, I had no illusions that any student would do in about 45 minutes what had ended up taking me several days to plot! The fact that so many students had recognized the issues and suggested ways to deal with them was very refreshing.

Remember the Terry Mantras? "Keep it simple" and "There is always a better and neater solution". How can one improve on this system, because it must be possible? What have I forgotten? (I have deliberately "forgotten" something - can you spot it? There might be a reward offered…)

As you probably now realize, the difficulties in the final implementation lie not with generating code, but with writing down a decent, self-consistent set of criteria.

Section C

Candidates were provided with some basic ideas, and asked to develop a version of a system that could transform specifications like the following:

```
Humanities
  FirstYear :
    Exclude : Total_Credits < 2 and (Average < 40.0 or DPR > 2);
              Lowest_Mark < 10.
    Merit_List :
              Average > 75 and Lowest_Mark > 60 and Year_Firsts >= 2 and Fails = 0.
  ThirdYear :
    AP_Complete_In_One :
              Total_Credits /= 10.0 and Years_Here > 3.
  AnyYear :
    Congratulate :
              Weighted_Average > 90.0 and Fails = 0.
```

into C# or Java code on the lines of the following:

```
class ActionSetBuilder {

  public static void BuildSet(Student s, OutFile logFile) {
  // Build up a set of numeric tags for Student s based on an analysis of his or her record
    s.actions = new IntSet();
    if (s.academicYear == 1) {
      if (s.fullTotalCredits < 2 && (s.rawAverage < 40.0 || s.DPRthisYear >= 2))
        s.actions.Incl(10);
      if (s.lowestMark < 10)
        s.actions.Incl(10);
      if (s.rawAverage > 75 && s.lowestMark > 60 && s.yearFirstsThisYear >= 2 && s.failsThisYear == 0)
        s.actions.Incl(16);
    }
    if (s.academicYear == 3) {
      if (s.fullTotalCredits < 10.0 && s.yearsOfStudy > 3)
        s.actions.Incl(21);
    }
    {
      if (s.weightedAverage > 90.0 && s.failsThisYear == 0)
        s.actions.Incl(11);
    }

  } // ActionSetBuilder.BuildSet

} // ActionSetBuilder
```

They were provided with an exam kit for Java or C#, containing the Coco/R system, along with a suite of simple, suggestive test specifications. They were told that later in the day some further ideas and hints would be provided.

Section D

(Summary of free information made available to the students 18 hours before the formal examination.)

To prepare themselves to answer Section B of the examination they were encouraged to study the attributed grammar in depth, and warned that the questions in Section B would probe this understanding, and that they might be called on to make some modifications and extensions.

Free information

Context-free unattributed LL(1) (grammar for a basic Converter

```
COMPILER Converter  $CN
/* Converter for generating a C# method for transforming a list of criteria for marking
   up student exam results from a simple specification into equivalent C# code.
   Context free grammar only.  Exam Kit provides an attributed version of this
   P.D. Terry, Rhodes University, 2013 */

IGNORECASE

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

TOKENS
  integer      = digit { digit } | digit { digit } CONTEXT (".") .
  double       = digit { digit } "." digit { digit }  .
  identifier   = letter { letter | digit | "_" ( letter | digit ) } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Converter    = Faculty
                 {   "FirstYear"  ":" CriteriaList
                   | "SecondYear" ":" CriteriaList
                   | "ThirdYear"  ":" CriteriaList
                   | "FourthYear" ":" CriteriaList
                   | "AnyYear"    ":" CriteriaList
                 } .

  Faculty      = Identifier .

  CriteriaList = { Action ":" Condition { ";" Condition } "." } .
  Action       = Identifier .

  Condition    = Expression .
  Expression   = AndExp { OrOp  AndExp } .
  AndExp       = EqlExp { AndOp EqlExp } .
  EqlExp       = RelExp { EqlOp RelExp } .
  RelExp       = AddExp [ RelOp AddExp ] .
  AddExp       = MulExp { AddOp MulExp } .
  MulExp       = Factor { MulOp Factor } .
  Factor       = Primary | ( "+" | "-" | NotOp ) Factor .
  Primary      = Designator | Double | Integer | "(" Expression ")" .
  Designator   = Identifier .

  Identifier   = identifier .
  Integer      = integer .
  Double       = double .
  AddOp        = "+" | "-" .
  MulOp        = "*" | "/" .
  EqlOp        = "=" | "/=" .
  RelOp        = "<" | "<=" | ">" | ">=" .
  NotOp        = "not" .
  AndOp        = "and" .
  OrOp         = "or" .

END Converter.
```