RHODES UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SUPPLEMENTARY EXAMINATIONS : JANUARY 2018

COMPUTER SCIENCE 301 - PAPER 2 - COMPILERS

Examiners:
Internal : Prof P.D. Terry
External : Prof M. Kuttel

Duration: 4 hours
Marks: 180
Pages: 19 (please check!)

The Concise Oxford English Dictionary may be used during this examination.

There are fourteen (14) questions. Answer ALL questions. Answers may be written in any medium except red ink, and are preferably written in the spaces provided on the question paper. You may use pencil, and you may also answer the questions by editing the supplied electronic copies of this material.

Hand in all material at the end of the examination.

A word of advice: The influential mathematician R.W. Hamming very aptly and succinctly professed that "the purpose of computing is insight, not numbers".

Several of the questions in this paper are designed to probe your insight - your depth of understanding of the important principles that you have studied in this course. If, as we hope, you have gained such insight, you should find that the answers to many questions take only a few lines of explanation. Please don't write long-winded answers.

Good luck!

(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 devise a Parva compiler targetting the Parva Virtual Machine interpreter system studied in the course. This compiler was to be extended to support simple string handling operations, missing from the original Parva compiler. Some 16 hours before the examination a complete grammar for such a compiler and other support files for building this system 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.)

DO NOT OPEN THIS PAPER UNTIL YOU ARE TOLD TO DO SO

Section A: Conventional questions [100 marks]

QUESTION A1 **[2 + 4 = 6 marks]**

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

(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. [4 marks]

QUESTION A2 **[6 marks]**

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

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

QUESTION A3 **[6 + 10 = 16 marks]**

(a) *Scope* and *Existence/Extent* are two terms that come up in any discussion of the implementation of block-structured languages. Briefly explain what these terms mean, and the difference between them. [6 marks]

(b) Scope rules for a simple block-structured language like Parva can be implemented by making use of a suitable data structure for the symbol table. Show what such a structure might look like when a top-down one-pass compiler reaches each of the points marked (1) and (2), if it compiles the program below. [10 marks]

```
void main () {
  int x = 10, y, z;
  while (x > 0) {
    read(y);
    bool isPosY = y > 0;
    int z = x / 2;
    x = x - y;
    // ++++++++++++++++++++++++++++ point (1)
  }
  int a = x + y + z;
  // ++++++++++++++++++++++++++++++ point (2)
}
```

QUESTION A4 **[15 + 15 = 30 marks]**

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

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

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

```
<Children> ::= ε  |  <Child> <Children>
<Child>    ::= David | Helen | Kayd | Jamie
```

The notation allowed the use of an explicit ε. () Parentheses, [] brackets and {} braces were not used. 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.

Using the same CHARACTERS as in the above grammar, give the TOKENS and PRODUCTIONS sections for a Cocol grammar that uses EBNF to describe BNF style productions (don't write a BNF style grammar to descripe EBNF style productions). Use the above grammar as a guide, use `"eps"` to represent ε and use BNF as the goal symbol. [15 marks]

(b) Assume that you have available a suitable scanner method called `GetSym` that can recognize the terminals of this *new* grammar for BNF and classify them appropriately as members of the following set:

```
{ EOFSym, noSym, EOLSym, termSym, nontermSym, definedBySym, epsilonSym, barSym }
```

Develop a hand-crafted recursive descent parser for recognizing a set of such BNF productions (not the original EBNF production set given earlier), that is 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]

QUESTION A5 **[2 + 2 + 6 + 6 = 16 marks]**

(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
```

Here are two attempts to write grammars describing an RPN expression:

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

and

```
(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]

(c) 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]

(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]

QUESTION A6 **[2 + 4 + 14 = 20 marks]**

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

(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]

(c) *(Static semantics)* 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 } ";"
```

So far the students have not been shown how to generate code, but have been made aware of the need for static semantic checking, as exemplified in the following simple production for an *Assignment* (for very simple assignments only):

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

Assuming that her Parva compiler will eventually support Boolean, integer and character types, what static semantic checks will be needed to implement the Python-like language extension, and will these entail any special techniques? The Cocol and C# code that is needed turns out to be quite intricate, so you need not attempt to give it in detail unless you wish to do so. Give an answer on the lines of

> Each of the Designators must ….
> ….

[14 marks]

QUESTION A7 **[3 + 3 = 6 marks]**

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

```
WhileStatement<StackFrame frame>      (. Label startLoop = new Label(known); .)
=  "while" "(" Condition ")"          (. Label loopExit = new Label(!known);
                                         CodeGen.BranchFalse(loopExit); .)
   Statement<frame>                   (. CodeGen.Branch(startLoop);
                                         loopExit.Here(); .) .
```

This generates code that matches the template

```
startLoop:    Condition
              BZE  loopExit
              Statement
              BRN  startLoop
loopExit:
```

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

```
whileLabel:   BRN  testLabel
loopLabel:    Statement
testLabel:    Condition
              BNZ  loopLabel
loopExit:
```

Their argument is that in most situations a loop body *is* executed many times, and that the efficiency of the system will be markedly improved by executing only one conditional branch instruction on each iteration.

(a) Do you think the claim is justified for an interpreted system such as we have used? Explain your reasoning. [3 marks]

(b) If the suggestion is easily implementable in terms of our code generating functions, show how this could be done. If it is not easily implementable, why not? [3 marks]

Section B [80 marks]

Please note that there is no obligation to produce a machine readable solution for this section. Coco/R and other files are provided so that you can enhance, refine, or test your solution if you desire. If you choose to produce a machine readable solution, you should create a working directory, unpack EXAM.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

Until yesterday, the Parva language that you have got to know so well had but three basic data types - integer, boolean, and character. Although it made provision for the use of string constants in `read` and `write` statement lists, it did not provide for the declaration and use of string variables:

```
void main() { $C+
  string yourName, myName, theBoff;
  myName = "Pat";
  theBoff = myName;
  read("What is your name? ", yourName);
  write(myName, " is pleased to meet ", yourName);
  if (myName != yourName)
    writeLine(" (Our names are different)");
  if (upper(myName) == upper(yourName))
    writeLine("(Our uppercased names are the same)");
} // main
```

Yesterday you made history when you were invited to develop a version of the Parva compiler that at last incorporated a `string` type, one which should have been able to support the following (at least):

(a) Declare variables and arrays of type `string`;
(b) Assign constant strings to such variables, and values of string variables to other string variables;
(c) Read and write values for string variables;
(d) Compare two strings for equality or inequality;
(e) Provide a function that returns an UPPERCASE version of a string;
(f) Perform any necessary semantic and constraint checking on strings.

Later in the day you were provided with a sample solution to that challenge. Continue now to answer the following unseen questions. **Where possible, express your answers in code rather than loose discussion.**

QUESTION B8 **[5 marks]**

The compiler as provided does not allow you to declare strings as named constants. Show (preferably give code) how this might be done, to allow, for example: [5 marks]

```
void main () {
  const title = "The thin edge of the wedge";
  string s = title;
} // main
```

QUESTION B9 **[10 marks]**

String handling libraries usually make provision for determining the length of a string. Show how to extend the compiler, code generator and interpreter to allow for Parva code like

```
void main () {
  string str;
  read(str);
  write(str, length(str) );
} // main
```

where the `length(str)` function can only be applied to a single parameter expression of `string` type. [10 marks]

QUESTION B10 **[10 marks]**

What does the system currently output if a program like the following is compiled and run?

```
void main () {
  string str;  // declared
  write(str);  // but never properly defined
} // main
```

Explain your reasoning. [10 marks]

QUESTION B11 **[15 marks]**

The situation in B10 is actually symptomatic of a larger problem. Some implementations of languages ensure that all variables not explicitly "initialised" at the point of creation are implicitly initialised to a known value - say zero - anyway. How could your compiler be changed to incorporate this device - both for simple variables and also for arrays (preferably give code)? [15 marks]

QUESTION B12 **[5 marks]**

Show how the system could be extended to report situations like that in B10 as runtime errors. [5 marks]

QUESTION B13 **[20 marks]**

The compiler currently allows for string comparisons for equality and inequality. Show the modifications that are needed to support "ordering" comparisons, such as exemplified by [20 marks]

```
void main () {
  string s1, s2;  // declared
  read(s1, s2);   // initialised
  if (s1 < s2)    // compare
    write(s1, " is lexically less than ", s2);
} // main
```

QUESTION B14 **[15 marks]**

String concatenation (joining) is another useful feature. Show how the compiler, code generator and interpreter should be modified to allow for a program like [15 marks]

```
void main () {
  string s1, s2;  // declared
  s1 = "start ";
  s2 = "end";     // initialised
  string joined = s1 + s2;
  write(joined);  // start end
} // main
```

END OF THE EXAMINATION QUESTIONS

Section C

(Summary of free information made available to the students 24 hours before the formal examination.)

Candidates were provided with the basic ideas, and were invited to extend a version of the Parva compiler to incorporate a `string` type.

It was pointed out that string types in languages that support them usually come with a complete support library of useful functions, but to begin with they might limit themselves to a system that would do the following:

(a) Declare variables and arrays of type `string`;
(b) Assign constant strings to such variables, and values of string variables to other string variables;
(c) Read and write values for strings;
(d) Compare two strings for equality or inequality;
(e) Provide a function that returns an UPPERCASE version of a string.

They were provided with an exam kit, containing the Coco/R system, a working Parva compiler that handles integer, character and boolean types, along with a suite of simple, suggestive test programs. 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 16 hours before the formal examination.)

A complete Parva compiler, incorporating one approach to the basic provision of a simple `string` type, was supplied to candidates in a later version of the examination kit. They were encouraged to study it in depth and warned that questions in Section B would probe this understanding; few hints were given as to what to expect, other than that they might be called on to comment on the solution, and perhaps to make some modifications and extensions. They were also encouraaged to spend some time thinking how any other ideas they had during the earlier part of the day would need modification to fit in with the solution kit presented to them.

END OF THE EXAMINATION PAPER