RHODES UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

EXAMINATIONS: NOVEMBER 2015

Computer Science 301

PAPER 2 - Translators

Internal Examiner: Prof P.D. Terry

MARKS: 180
DURATION: 4 hours

External Examiner: Prof P. Blignaut

GENERAL INSTRUCTIONS TO CANDIDATES

---

1. This paper consists of 10 pages and 15 questions. Please ensure that you have a complete paper.

2. You will be provided with a separate document that indicates the hand-in procedure for the examination.

Full marks may be obtained by answering questions 1 - 13.

You may answer either of both of 14 and 15 to earn bonus marks if you have time.

Answers may be written in any medium except red ink.

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 - as Einstein put it "Keep it as simple as you can, but no simpler".

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 develop graphics facilities for Parva. Some 16 hours before the examination a complete grammar and other support files for building such a 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 system, access to a computer, and machine readable copies of the questions.)

PLEASE DO NOT TURN OVER THIS PAGE UNTIL TOLD TO DO SO

Section A: Conventional questions [100 marks]

QUESTION A1 **[5 marks]**

A1 *Pragmas* and *comments* often appear in source code submitted to compilers. What property do pragmas and comments have in common, and what is the semantic difference between them? [4 marks]

QUESTION A2 **[6 marks]**

A2 What do you understand by the term *short circuit semantics* as applied to the evaluation of Boolean expressions? Illustrate your answer by means of some specimens of code that incorporate simple Boolean expressions, distinguishing between short-circuit semantics and some other alternative. [6 marks]

QUESTION A3 **[12 marks]**

A3 In the practicals of this course you experimented with a decompiler named dotPeek.

(a) What distinguishes a "compiler" from a "decompiler"? [2 marks]

(b) Draw a T-diagram that captures the essence of the dotPeek decompiler. [2 marks]

(c) The developers of dotPeek may well have used C# in the development process. Given that they had a C# compiler like `csc.exe` that could run on .NET, draw T-diagrams illustrating how the final version of dotPeek (which you ran on .NET) might have been produced. [3 marks]

(d) Using further T-diagrams, suggest and explain how the developers of dotPeek might have performed a self-consistency test of their product before releasing it to users. [5 marks]

A set of blank T-diagrams is provided as an addendum to this paper, which you can complete and submit with your answer book.

QUESTION A4 **[15 marks]**

A4 Consider the following Cocol specification:

```
COMPILER Expressions

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

TOKENS
  ident  = letter { letter | digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Expressions  = { Term "?" } EOF .
  Term         = Factor { "+" Factor | "-" Factor } .
  Factor       = ident [ "++" | "--" ] | [ "abs" ] "(" Term ")" .
END Declarations.
```

Develop a handcrafted scanner (not a parser!) that will recognize the tokens used in this grammar. Assume that the first few lines of the scanner routine are introduced as

```
static int getSym() {
  while (ch != -1 && ch <= ' ') getChar();
```

and that the `getSym()` method must return an integer representing the token that it has recognized, as one of the values that you can denote by the names:

```
plus, minus, plusPlus, minusMinus, query, lParen, rParen, abs, ident, error, EOF
```

Assume that when the scanner invokes the source handling method

```
static void getChar()
```

this will assign to a static field `ch` (of the class of which these methods are members) the next available character in the source text, or -1 if no more characters can be read. [15 marks]

QUESTION A5 **[42 marks]**

A5 Consider the following grammar, expressed in EBNF, that describes the form of a typical university course:

```
Course       = Introduction Section { Section } Conclusion .
Introduction = "lecture" [ "handout" ] .
Section      = { "lecture" | "prac" "test" | "tutorial" | "handout" } "test" .
Conclusion   = [ "panic" ] "Examination" .
```

(a) What do you understand by the statement "two grammars are equivalent"? [2 marks]

(b) What do you understand by the statement "a grammar is ambiguous"? [2 marks]

(c) Rewrite these productions so as to produce an equivalent grammar in which no use is made of the EBNF meta-brackets { … } or […]. [8 marks]

(d) Clearly explain why the derived set of productions does not obey the LL(1) constraints. [4 marks]

(e) It is suspected that the original grammar might be ambiguous. Give, with reasons, an example of a sentence that would confirm this suspicion. [4 marks]

(f) Why does it not follow that every grammar that is not LL(1) must be ambiguous? Justify your argument by giving a simple example of a non-LL(1) grammar that is not ambiguous. [4 marks]

(g) Does it follow that if a recursive descent parser is constructed for this grammar, it is doomed to fail? Justify your answer. [3 marks]

(h) The University Senate have decreed that courses may no longer be offered if they do not obey various constraints. In particular, an acceptable course may not

(1) offer fewer than 50 lectures, or subject the candidates to more than 8 tests;
(2) hold a practical until at least 4 lectures have been given;
(3) hold more practicals than tutorials.

Show how the grammar may be augmented to impose these constraints. A spaced copy of the grammar appears in the addendum, which you are invited to complete and hand in with your answer book. [15 marks]

QUESTION A6 **[20 marks]**

A6 Generations of Parva programmers have complained about the absence of a *for* loop, and it has been decided to add this feature, using syntax suggested by Pascal, and exemplified by

```
for i = 1 to 10 write(i);

for j = i + 1 to i - 4 {
  read(i); total = total + i;
}
```

(a) Write an EBNF description that should allow you to recognize such a statement, taking care not to be over-restrictive. [2 marks]

(b) What static semantic constraints must be satisfied by the various components of your production(s)? You might illustrate this by writing the Cocol attributes; alternatively just specify them in concise English. [4 marks]

(c) Critically examine and comment in detail on the suggestion that a *for* loop of the form

```
for i = start to stop {
  // something
}
```

should be regarded as (dynamically) semantically exactly equivalent to the following: [8 marks]

```
i = start;
while (i <= stop) {
  // something
  i++;
}
```

(d) If a *for* loop is implemented as in (c) - or, indeed, in any other way - problems would arise if the code indicated by "something" were to alter the value of the control variable. Do you suppose a compiler could forbid this practice? If so, how - and if not, why not? [6 marks]

Section B: Parva with simple graphics [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.

Yesterday you made history when you were invited to develop an extended Parva compiler which would provide support for simple graphics. Later in the day you were provided with a sample solution to that challenge, and the files needed to build that system have been provided to you again today. Time to ask a few searching questions!

Your answers should, wherever possible, be expressed in "code terms", and not vague statements such as "add a Boolean parameter to the production". To illustrate, suppose you were asked the question "How would you extend the compiler to allow for statements like `x++`; or `list[i]--`;?" An excellent answer would be:

Such statements are a convenient form of assignment statement, so change the `AssignmentOrCall` *production as shown below. The Code Generator and PVM already support the* `INC` *opcode and need no further changes.*

```
  AssignmentOrCall                 (. int expType;
                                      DesType des;
                                      string name;
                                      bool inc = true; .)
  = (  (  IF (IsCall(out des))     // use resolver to handle LL(1) conflict
          Ident<out name>          (. CodeGen.FrameHeader(); .)
          "("  Arguments<des>      (. CodeGen.Call(des.entry.entryPoint); .)
          ")"
        | Designator<out des>      (. if (des.entry.kind != Kinds.Var)
                                        SemError("cannot assign to " + Kinds.kindNames[des.entry.kind]); .)
***       ( AssignOp
            Expression<out expType> (. if (!Compatible(des.type, expType))
                                       SemError("incompatible types in assignment");
                                       CodeGen.Assign(des.type); .)
***         | ( "++" | "--"          (. inc = false; .)
***           )                      (. if (!IsArith(des.type))
***                                    SemError("arithmetic type needed");
***                                    CodeGen.IncOrDec(inc); .)
***       )
        )
      | ( "++" | "--"                (. inc = false; .)
        ) Designator<out des>        (. if (des.entry.kind != Kinds.Var)
                                        SemError("variable designator required");
                                       if (!IsArith(des.type))
                                        SemError("arithmetic type needed");
                                       CodeGen.IncOrDec(inc); .)
    ) WEAK ";" .
```

Continue now to answer the following unseen questions:

QUESTION B7 **[4 marks]**

B7 The $ST, $SD and $HD pragmas are very useful, but a bit irritating to a simple user. How can the system be modified so that they only take effect if the $D debugging pragma has been turned "on"? [4 marks]

QUESTION B8 **[4 marks]**

B8 The syntax of the main `Parva` production is essentially:

Parva = { FuncDeclaration } EOF .

Is any practical advantage gained by the addition of the explicit EOF? Would the compiler not perform as well if EOF were to be omitted? Discuss briefly. [4 marks]

QUESTION B9 **[10 marks]**

B9 "Anyone can write a compiler for error-free input". The system provided to you was clearly written in a hurry, and several checks have simply been omitted in the parts dealing with parameters for the Turtle Graphics statements. Identify some of these and suggest what semantic checking should be added, and where this must occur. [10 marks]

QUESTION B10 **[8 marks]**

B10 The Turtle Graphics commands have been restricted to `Forward` and `TurnLeft`. Can the Parva system be modified to allow obvious extensions `Backward` and `TurnRight`, *without adding* to the PVM and Code Generator classes? If so, how, and if not, why not? [8 marks]

QUESTION B11 **[6 marks]**

B11 Two students were overheard comparing possible ways of implementing the *if-then-else* construct in Parva. One of them had developed the solution suggested in the kit:

```
IfStatement<StackFrame frame>     (. Label falseLabel = new Label(!known);
                                     Label outLabel = new Label(!known); .)
=  "if" "(" Condition ")"         (. CodeGen.BranchFalse(falseLabel); .)
      Statement<frame>
   (   "else"                     (. CodeGen.Branch(outLabel);
                                     falseLabel.Here(); .)
         Statement<frame>         (. outLabel.Here(); .)
     | /* no else part */          (. falseLabel.Here(); .)
 )
 .
```

while the other had come up with something slightly different:

```
IfStatement<StackFrame frame>     (. Label falseLabel = new Label(!known);
                                     Label outLabel = new Label(!known); .)
=  "if" "(" Condition ")"         (. CodeGen.BranchFalse(falseLabel); .)
      Statement<frame>            (. CodeGen.Branch(outLabel);
                                     falseLabel.Here(); .)
   [ "else" Statement<frame> ]    (. outLabel.Here(); .)
.
```

Which of these "works"? Perhaps both "work"? In that case, which, if either, is better than the other, and why? [6 marks]

QUESTION B12 **[24 marks]**

B12 Here are two trivial programs that will currently compile without complaining but must surely misbehave. How should the system be improved to handle this? Can problems like this be anticipated and detected at compile time? If so, how? If not, could an improved implementation of the PVM help? Show how this could be achieved. [24 marks]

```
void Main() {
  DrawLine(0, 0, 50, 50);
} // Main

void Main() {
  InitGraphics(500, 500);
  DrawLine(0, 0, 50, 50);
  CloseGraphics();
  DrawLine(50, 50, 0, 100);
} // Main
```

QUESTION B13 **[24 marks]**

B13 The default location and orientation of the axes on the graphics canvas is bound to confuse people more familiar with axes where the y-axis points upwards. As a rather more interesting challenge, suppose the `InitGraphics` method were also to allow an alternative: `InitGraphics(width, height, x0, y0)`, whose effect would be to open a canvas (as before), but in this case arrange that subsequent calls to `Line(x1, y1, x2, y2)` would draw the specified line on a canvas whose **centre** corresponds to the point (`x0, y0`) and where the x- and y-axes pointed **right** and **up**. If the third and fourth arguments `x0` and `y0` are omitted, they should be taken to have the values `(0, 0)`.

This is not as difficult as it might at first appear, once it is realised that a simple linear transformation within the PVM of the values passed as arguments to `Line()` will produce the desired effect. To save you a lot of time, the required transformations have been explained in detail on page 8.

Modify the grammar and the PVM to accommodate the revised semantics of `InitGraphics`. [24 marks]

QUESTION B14 (Bonus - optional) **[8 marks]**

B14 If, as suggested yesterday, you execute the sample programs in the kit using the system provided to you today you should notice that, while some "work" very well, some others - deceptively simple - seem to perform very badly (for example EG09, EG13 and EG15). Why do you suppose this is? Be assured that it is not the turtle logic that is incorrect; it must be something else. Can you suggest a fairly simple way to improve the system? (Give code!) [8 marks]

QUESTION B15 (Bonus - optional) **[8 marks]**

B15 Consider the program below. In C# and Parva, if a parameter is to be passed "by reference", the keyword `ref` is used to mark both the formal parameter and the corresponding actual argument, as shown in the first function call. In C, C++, Pascal and Modula-2 and various other languages, the equivalent of `ref` is used only on the formal parameter (as illustrated by the second function call).

Discuss whether it would be possible and/or preferable to use the "C" convention in Parva. If it is possible, explain how it could be done (code!); if it is not possible, explain why not. [8 marks]

```
void Interchange(ref int x, ref int y) {
  int z = x; x = y; y = z;
} // Interchange

void Main() {
  int a = 56, b = 63;
  Interchange(ref a, ref b);   // C# and Parva style
  Interchange(a, b);           // C, Pascal, Modula-2 style
} // Main
```

END OF 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 for adding simple graphics primitives to the Parva language, including support for Turtle Graphics, and were invited to modify a supplied Parva implementation to accommodate these.

They were provided with an exam kit for C#, containing a working Parva compiler like that which they had used in the practical course. They were also given a suite of simple, suggestive test programs, and some snapshots of the graphics output from these examples. Finally, 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.)

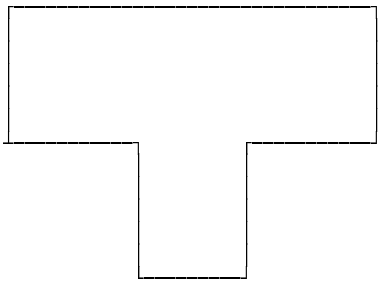
A complete Parva system incorporating the features they had been asked to implement was supplied to candidates in a later version of the examination kit. They were encouraged to study it in depth and warned that questions in Section B would probe this understanding; few hints were given as to what to 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 encouraged 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. The system as supplied at this point was deliberately naïve in some respects, in order to lay the ground for the unseen questions of the following day.

(This diagram should help to explain Question B13)

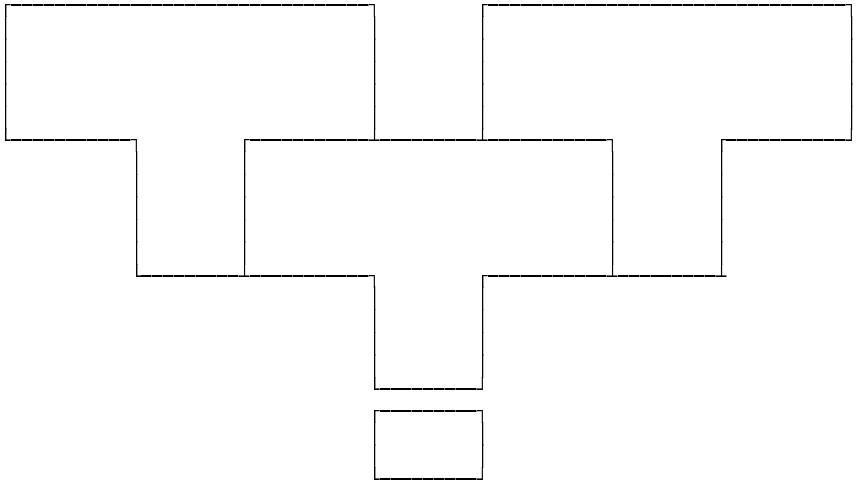
ADDENDUM 1 (Question A3) - Fill in your Student number

T Diagrams for explaining the development of dotPeek.

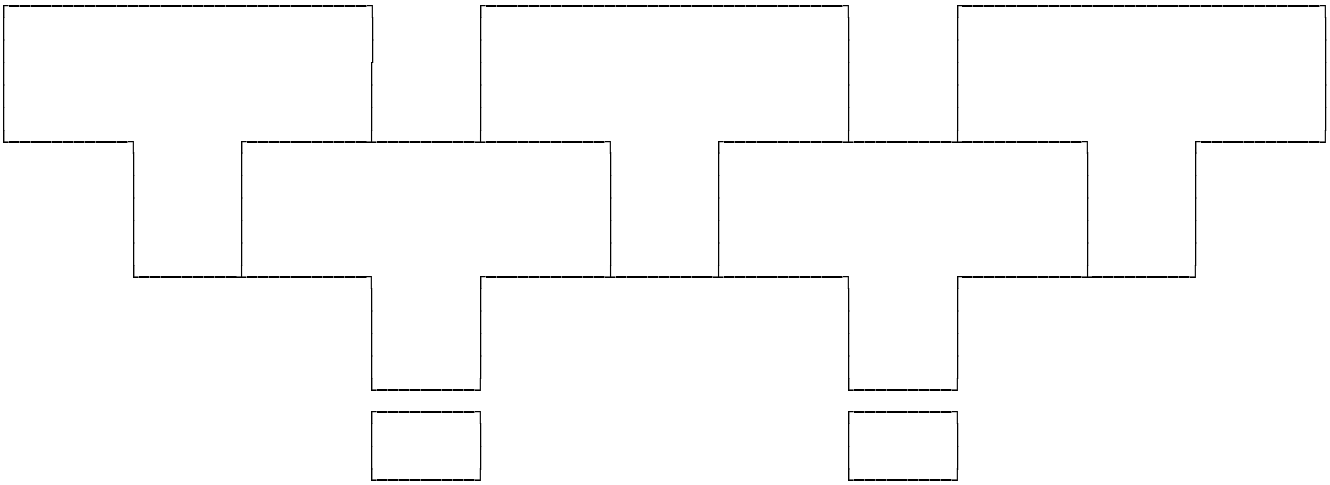
DotPeek itself



Developing DotPeek



Self-consistency check



ADDENDUM 2 (Question A5) - Fill in your Student number

```
COMPILER Course


IGNORE CHR(0) .. CHR(31)
PRODUCTIONS

  Course
  = Introduction

    Section

    { Section

    } Conclusion



  .

  Introduction
  = "lecture"

    [ "handout"
    ] .


  Section
  = {   "lecture"

      | "prac"


        "test"


      | "tutorial"

      | "handout"

    }

    "test"


  .


  Conclusion
  = [ "panic"
    ]

    "Examination"
  .

END .
```