

RHODES UNIVERSITY
November Examinations - 2008
Computer Science 301 - Paper 2 (Solutions)

Examiners:
Prof P.D. Terry
Prof S. Berman

Time 3 hours
Marks 180
Pages 12 (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 develop a Forth-like assembler for the PVM. 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.)

Section A [100 marks]

A1. (a) What distinguishes a "compiler" from an "assembler"? [2]

A compiler translates "high level" code to object/machine code; an assembler translates low-level code to object/machine code.

(b) How would you explain and distinguish the terms "self-resident compiler", "cross-compiler" and "self-compiling compiler" to a student taking a course in compiler construction for the first time, and how could you lead them to believe that a self-compiling compiler can ever be a reality? [8]

Self-resident - the compiler runs on the same machines as it generates code for.

Cross-compiler - the compiler generates code for a different machine from the one on which it executes.

Self-compiling compiler - source for the compiler is available in the language that is to be compiled. This implies that the compiler should be able to replicate its own object/machine code. Development of such compilers requires that they first be implemented in some other host language for which a compiler already exists, and are then hand-translated into a version written in the source code (itself a classical bootstrap situation).

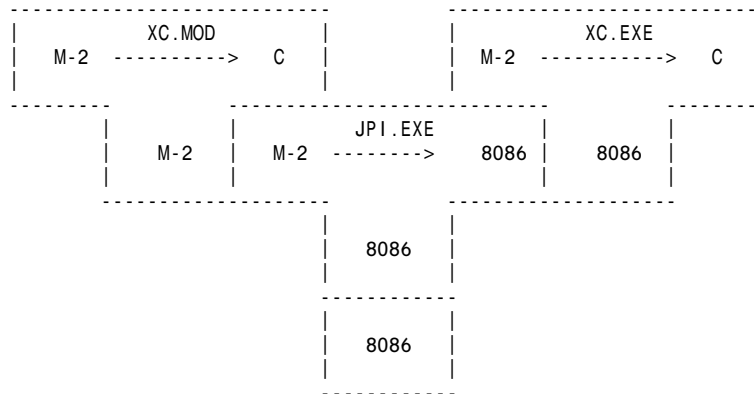
A2. Explain clearly and simply what you understand by the terms syntax, static semantics, and dynamic semantics, and what distinguishes one from another. [6]

*Syntax rules describe the arrangement or form that program source, statements and expressions can take or in which they can be combined (for example, in Java source code a while statement requires a parenthesized expression and another statement to follow the parentheses). Static semantic rules relate to the way in which types, expressions identifiers and so on interact (for example, in a while statement the parenthesized expression must be one composed of operands and operators that not only satisfy syntactic rules, but which also evaluates to a Boolean result). Dynamic semantics describe what happens when the program (or statement) executes (for example, in the case of a while statement the subsidiary statement will be executed, possibly repeatedly, for as long as a re-evaluation of the expression yields a Boolean value of **true**).*

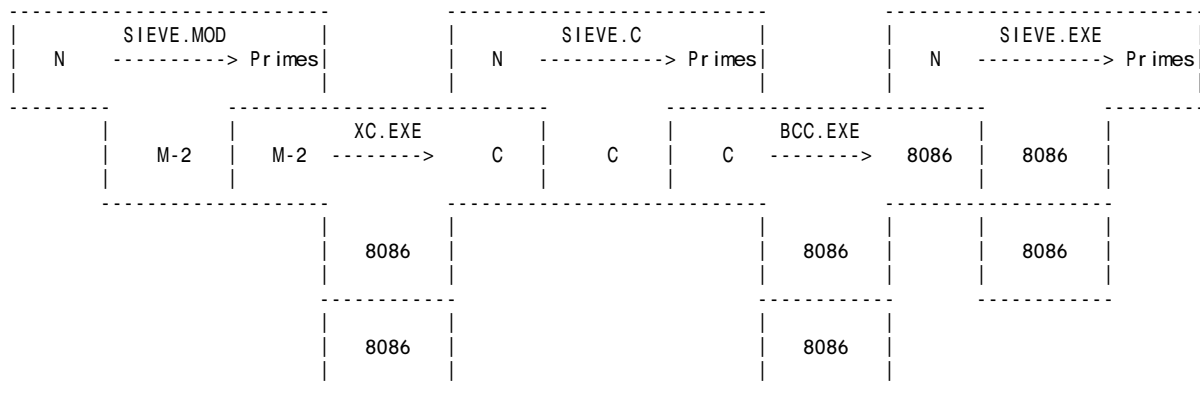
A3. In the practical sessions you should have used the Extacy Modula-2 to C translator. This was developed in Russia by a team who used the JPI Modula-2 compiler available for the PC. The demonstration system we downloaded from the Internet came with the file XC.EXE, and a few other modules written in Modula-2 (but not the source of the XC.EXE executable itself). Draw T-diagrams showing the process the Russians must have used to produce this system, and go on to draw T-diagrams showing how you managed to take a program SIEVE.MOD and run it on the PC using the Borland C++ system as your compiler of choice. [10]

A set of blank T-diagrams is provided in the free information, which you can complete and submit with your answer book.

The XC program was written in Modula-2 and compiled:



Using XC.EXE can be depicted as follows:



- A4. (a) What would you understand by a statement from a group of very unhappy students who came to you and told you that they had discovered that their carefully thought out grammar had turned out to be "ambiguous"? [2]

An ambiguous grammar is one where at least one sentence of the language it describes can be derived in more than one way, that is, one can find more than one parse tree representing that sentence

- (b) Why can an LL(1) conformant grammar never be ambiguous? (Explain carefully!) [3]

Because an LL(1) grammar has the property that parsing can only proceed on the basis of a single look ahead, and this path will thus always be uniquely defined.

- (c) Does it then follow that if a grammar is not ambiguous it must automatically be LL(1) conformant? Support your argument by giving some simple example grammars. [5]

No it does not. A typical example of a non-LL(1), unambiguous grammar is the classic left recursive one for simple expressions

```

Goal      = Expression .
Expression = Term | Expression "-" Term | Expression "+" Term .
Term       = Factor | Term "*" Factor | Term "/" Factor .
Factor     = "a" | "b" | "c" | "d" .

```

- (d) Discuss the anomaly in programming language design that gives rise to the so-called "dangling else" problem. Show how one could easily design a language that does not suffer from this problem. Also explain why the problem is in any case less severe than it might at first appear. [6]

The "dangling else" is the ambiguity that arises when one wishes to define the if-else statement as

IfElseStatement = "if" "(" Condition ")" Statement ["else" Statement] .

which allows a construct like

`"if" (boolExp1) "if" (boolExp2) statement1 "else" statement2`

to be derived in two different ways, corresponding either to

`"if" (boolExp1) { "if" (boolExp2) statement1 } "else" statement2`

or

`"if" (boolExp1) { "if" (boolExp2) statement1 "else" statement2 }`

The problem is easily eliminated if one insists on a closing keyword:

`IfElseStatement = "if" "(" Condition ")" Statement ["else" Statement] "endif".`

But it is not really a problem, as a recursive descent parser will naturally bind the "else" clause to the most recent "if" and produce the semantic effect that is needed.

A5. The Cocol grammar below attempts to describe declarations in a simple C-like language:

```

COMPILER Declarations

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

TOKENS
  ident = letter { letter | digit | "_" ( letter | digit ) } .

COMMENTS FROM "/*" TO "*/"

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Declarations = { DeclList } EOF .
  DeclList    = Type OneDecl { "," OneDecl } ";" .
  Type        = "int" | "void" | "bool" | "char" .
  OneDecl     = "*" OneDecl | Direct .
  Direct      = ( ident | "(" OneDecl ")" ) [ Params ] .
  Params      = "(" [ OneParam { "," OneParam } ] ")" .
  OneParam    = Type [ OneDecl ] .
END Declarations

```

(a) Is this an LL(1) compliant grammar? Explain your reasoning in some detail. [6]

Yes it is. The traditional full analysis might proceed by transforming the grammar to one with no meta-brackets. This is straightforward but a bit tedious:

```

Declarations      = DeclarationSeq EOF .
DeclarationSeq    = DeclList DeclarationsSeq | .
DeclList          = Type OneDecl MoreDecls ";" .
Type              = "int" | "void" | "bool" | "char" .
OneDecl           = "*" OneDecl | Direct .
MoreDecls         = "," OneDecl MoreDecls | .
Direct            = ( ident | "(" OneDecl ")" ) OptParams .
OptParams         = "(" ParamList ")" | .
ParamList         = OneParam MoreParams | .
MoreParams        = "," OneParam MoreParams | .
OneParam          = Type OptDecl .
OptDecl           = OneDecl | .

```

For Rule 1 we have only to consider the productions for Type and for Direct, which clearly cause no trouble. For Rule 2 we discern that the nullable non-terminals now are

DeclarationsSeq, MoreDecls, OptParams, ParamList, MoreParams, OptDecl

but these cause no trouble, as can be seen from their FIRST and FOLLOW sets

```

DeclarationsSeq
first:  "int" "void" "bool" "char"    follow: EOF

MoreDecls
first:  ", "                          follow:  "; "

OptParams
first:  "("                          follow:  "; " ", " ")"

ParamList
first:  "int" "void" "bool" "char"    follow:  ")"

MoreParams
first:  ", "                          follow:  ")"

OptDecl
first:  ident "*" "("                follow:  "; " ", " ")"

Direct
first:  ident "("                    follow:  "; " ", " ")"

OneParam
first:  "int" "void" "bool" "char"    follow:  "; " ", " ")"

```

However, this grammar is just as easily analysed in the original EBNF form

```

Declarations = { DeclList } EOF .
DeclList     = Type OneDecl { "; " OneDecl } ";" .
Type         = "int" | "void" | "bool" | "char" .
OneDecl      = "*" OneDecl | Direct .
Direct       = ( ident | "(" OneDecl ")" ) [ Params ] .
Params       = "(" [ OneParam { "; " OneParam } ] ")" .
OneParam     = Type [ OneDecl ] .

```

For Rule 1 we have only to consider the productions for Type and for Direct, which clearly cause no trouble. For Rule 2 we consider the nullable portions, which are

```

{ DeclList } and [ Params ] and [ OneParam { "; " OneParam } ]
and { "; " OneDecl } and [ OneDecl ]

FIRST( {DeclList} ) = FIRST(Type) = { "int", "void", "bool", "char" }
FOLLOW( {DeclList} ) = { EOF }

FIRST( [ Params ] ) = { "(" }
FOLLOW( [ Params ] ) = FOLLOW(Direct) = FOLLOW(OneDecl) = { "; " ", " ")" }

FIRST( [ OneParam { "; " OneParam } ] ) = FIRST(OneParam) = FIRST(Type)
FOLLOW( [ OneParam { "; " OneParam } ] ) = { ")" }

FIRST( { "; " OneParam } ) = { "; " }
FOLLOW( { "; " OneParam } ) = { ")" }

FIRST( { "; " OneDecl } ) = { "; " }
FOLLOW( { "; " OneDecl } ) = { ";" }

FIRST( [ OneDecl ] ) = { "*", ident "(" }
FOLLOW( [ OneDecl ] ) = { FOLLOW( OneParam ) = { "; " ", " ")" } .

```

and these clearly satisfy Rule 2.

- (b) Assume that you have accept and abort routines like those you used in this course, and a scanner `getSym()` that can recognise tokens that might be described by the enumeration

```

EOFSym, noSym, identSym, intSym, voidSym, boolSym, charSym, commaSym, lparenSym,
rparenSym, semicolonSym, starSym;

```

How would you complete the parser routines below? [20]

A spaced copy of this system appears in the free information, which you are invited to complete and hand in with your answer book.

This is straightforward but a bit time-consuming. Students have seen several examples like this, and generally master them pretty well

```
static SymSet
  FirstDeclarations = new SymSet(intSym, voidSym, boolSym, charSym),
  FirstDeclList     = new SymSet(intSym, voidSym, boolSym, charSym),
  FirstType         = new SymSet(intSym, voidSym, boolSym, charSym),
  FirstOneDecl      = new SymSet(starSym, identSym, lparenSym),
  FirstDirect       = new SymSet(identSym, lparenSym),
  FirstParams       = new SymSet(lparenSym),
  FirstOneParam     = new SymSet(intSym, voidSym, boolSym, charSym),

static void Declarations () {
  // Declarations = { DeclList } EOF .
  while FirstDeclList.contains(sym.kind) DeclList();
  accept(EofSym, "EOF expected");
}

static void DeclList () {
  // DeclList = Type OneDecl { "," OneDecl } ";" .
  Type();
  OneDecl();
  while (sym.kind == commaSym) {
    getSym(); OneDecl();
  }
  accept(semiColonSym, "; expected");
}

static void Type () {
  // Type = "int" | "void" | "bool" | "char" .
  if FirstType.contains(sym.kind) getSym();
  else abort("invalid type");
}

static void OneDecl () {
  // OneDecl = "*" OneDecl | Direct .
  if (sym.kind == starSym) {
    getSym(); oneDecl();
  }
  else if (FirstDirect.contains(sym.kind)) Direct();
  else abort("invalid start to OneDecl");
}

static void Direct () {
  // Direct = ( ident | "(" OneDecl ")" ) [ Params ] .
  if (sym.kind == identSym) getSym();
  else if (sym.kind == lparenSym) {
    getSym();
    OneDecl();
    accept(rparenSym, ") expected");
  }
  else abort("invalid start to Direct");
  if (sym.kind == lparenSym) Params();
}

static void Params () {
  // Params = "(" [ OneParam { "," OneParam } ] ")" .
  accept(lparenSym, "(" expected");
  if (FirstOneParam.contains(sym.kind)) {
    OneParam();
    while (sym.kind == commaSym) {
      getSym(); OneParam();
    }
  }
  accept(rparenSym, ")" expected");
}

static void OneParam () {
  // OneParam = Type [ OneDecl ] .
  Type();
  if (FirstOneDecl.contains(sym.kind)) OneDecl();
}
```

- A6. The examination results for a class of students are to be supplied in a file which gives, for each student, their student number, surname, gender, faculty, and mark obtained - a typical extract might read

63T0844	Terry	Male	S	85
06M1234	MacDonald	Female	H	50.00
06O1234	O'Malley	Male	C	78.6
06S1234	Smith-Jenkins	Male	S	55.67
81W4251	Bradshaw	Female	S	99
02F4567	Van Wyk Smith	Female	C	48
01M1234	McGregor	Male	L	10.30

Student numbers are of a standard format - two digits giving the year of first registration (02 denotes 2002, 95 denotes 1995 and so on), followed by the initial letter of the surname at the time of registration and then by a sequence of four digits. Names begin with a capital letter, and composite surnames like "Van Wyk Smith" and ones with embedded apostrophes and hyphens are also permissible. The faculty is denoted by a single letter with an obvious relationship to our faculties of Science, Humanities, Commerce, Law, Education and Pharmacy.

- (a) Show how an attributed Cocol grammar could be used to parse such a file and determine the number of students in the list who first registered more than three years ago. A skeleton grammar file is provided in the free information, which you could complete and submit with your answer book. [15]

Again, this is pretty straightforward and students have had experience of several similar grammars. The sort of solution I expected to receive is as follows (a Java version):

```
import Library.*;

COMPILER Marks $NC /* token names, generate compiler driver */
/* P.D. Terry, Rhodes University, 2008 */

static int
    total = 0,
    longer = 0;

CHARACTERS
    lf      = CHR(10) .
    control = CHR(0) .. CHR(31) .
    letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
    uletter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
    digit   = "0123456789" .
    printable = ANY - control .

TOKENS
    name      = uletter { "'" uletter | "-" uletter | letter } .
    studentNumber = digit digit uletter digit digit digit .
    mark      = digit { digit } [ "." digit { digit } ] .
    eol       = lf .

IGNORE CHR(9) + CHR(11) .. CHR(13)

PRODUCTIONS

Marks
= {
    StudentNumber
    FullName
    Gender
    Faculty
    mark SYNC eol    (. total++; .)
}                    (. IO.writeLine("Total students: " + total);
                    IO.writeLine(longer +
                                " of these first registered more than 3 years ago"); .)

.

FullName
= name { name } .

Gender
= "Male" | "Female" .

Faculty
= "S" | "H" | "C" | "E" | "L" | "P" .
```

```

StudentNumber    (. int year; .)
= studentNumber  (. String s = token.val.substring(0, 2);
                  try {
                      year = Integer.parseInt(s);
                  } catch (NumberFormatException e) {
                      year = 0; SemError("number too large");
                  }
                  if (year <= 8) year = 2000 + year; else year = 1900 + year;
                  if (2008 - year >= 3) longer++; .)

```

END Marks.

However, a submission from a student suggested that one only really needs to look at the first two characters on each line. Correctly developed this is reflected in code like the following (a C# version). (As it happened the student's submission was incorrect - he forgot that one would still have to read the superfluous material on each line, which is why the `eof` token has a rather odd definition).

```

using Library;

COMPILER Marks1 $NC /* token names, generate compiler driver */
/* P.D. Terry, Rhodes University, 2008 */

static int
total = 0,
longer = 0;

CHARACTERS
lf          = CHR(10) .
cr          = CHR(13) .
control     = CHR(0) .. CHR(31) .
uletter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
digit       = "0123456789" .
printable   = ANY - control .

TOKENS
studentNumber = digit digit .
eof           = uletter { printable } [ cr ] lf .

IGNORE CHR(9) + CHR(11) .. CHR(13)

PRODUCTIONS

Marks1
= { studentNumber    (. int year;
                      try {
                          year = Convert.ToInt32(token.val);
                      } catch (Exception) {
                          year = 0; SemError("invalid number");
                      }
                      if (year <= 8) year = 2000 + year; else year = 1900 + year;
                      if (2008 - year >= 3) longer++; .)

    SYNC eof         (. total++; .)
}                    (. IO.WriteLine("Total students: " + total);
                      IO.WriteLine(longer + " of these first registered more than 3 years ago"); .)

END Marks1.

```

- (b) Clearly this problem can be solved very easily without the use of a tool like Coco/R. Are there any advantages to be gained from using Coco in simple applications of this sort, and if so, what might these advantages be? [5]

The advantages of using Coco are that the parser and scanner are generated very easily from a rigorous specification, and the parser can also incorporate error checking very easily rather than requiring a programmer to hack away (or more than likely, duck error checking completely!)

- A7. Suppose that you are developing 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 parser for a non-terminal A of the grammar G so as to incorporate simple but effective error recovery. [12]

Standard bookwork as discussed in the text book. But, disappointingly, few had read this!

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.

Yesterday you made history when you were invited to develop a basic assembler for generating PVM code from assembler source inspired by the Forth language.

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. Continue now to answer the following unseen questions.

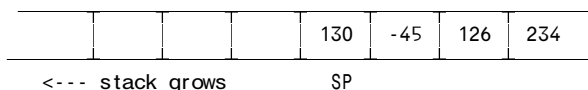
- B8. At present the system will allow a user to redeclare another variable or constant using an existing name, or even to define a variable or constant that has the same name as a word like NEGATE. While this might be thought to be useful, it is also dangerous. How would the system need to be modified to prevent such behaviour? [5]

This is actually pretty trivial - we change the grammar to include the check:

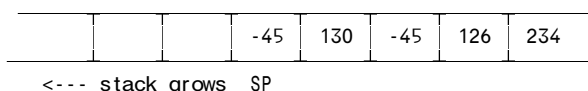
```
Ident<out string str>
= wordToken
    (. str = token.val.ToUpper();
    bool okay = Char.IsLetter(str[0]);
    int i = 0;
    while (okay && i < str.Length) {
        okay = okay && Char.IsLetterOrDigit(str[i]);
        i++;
    }
    if (!okay) SemError("Invalid identifier");
    if (Dictionary.FindWord(str) != null) // ++++++
        SemError("Identifier redeclared"); .)
```

- B9. The language Forth had a number of other words that were of use in writing programs that manipulated the stack efficiently. Among these were DROP, which simply discarded the top element from the stack, SWAP, which had the effect of exchanging the two items at the top end of the stack, and OVER, which had the effect of pushing another copy of the item just below the top of stack onto the stack. Show how these new words could be added to the system, by introducing new opcodes to the PVM and corresponding entries to the standard dictionary. [15]

Hint: As an example of using OVER, if the stack originally looked like this:



then after executing OVER it should look like



These are very easy and are supposed to be confidence boosters. Students had often made small additions to the PVM in practicals (although they have not seen these ones)


```

case PVM.drop:           // discard tos
    pop();
    break;
case PVM.swap:           // exchange top two elements on stack
    tos = pop();
    sos = pop();
    push(tos);
    push(sos);
    break;
case PVM.over:           // push (copy) old sos above tos
    tos = pop();
    sos = pop();
    push(sos);
    push(tos);
    push(sos);
    break;

```

- B10. Given the introduction of these words, explain what the effect would be of the following sequence of code: [10]

```

A @ B @
OVER OVER < IF SWAP ENDIF DROP
.

```

This may be too much to expect, under pressure?

It pushes the values of variables A and B, and then replicates these so that the stack has values

A B A B

The < compares the top two copies and leaves the truth value of the comparison $A < B$ on top of the stack.

A B A < B

If $A < B$ we then swap the original A and B - leading to B A, If not we are led to A B. Dropping the top of stack after this leaves the original B on the stack if $A < B$, or A on the stack if $A > B$ - in other words this is a version of a "max" function.

- B11. Show how one could add an ELSE clause to the IF ... ENDIF construction, thus permitting one to write code of the form

```
IF "correct" ELSE "incorrect" ENDIF
```

to display an appropriate message based on the value currently on the top of the stack. [10]

Devoid of any error checking this would become a fairly simple addition to the system. We add the ELSE word to the dictionary in the obvious way and then have a code generating method as shown below (many submissions omitted to generate a branch instruction, or generated a second branchfalse instruction for reasons that were not clear to me. Exam nerves perhaps?)

```

public static void codeForElse() {
    Label falseCode = labels.pop();
    labels.push(new Label(!known)); // endif exit
    branch(labels.peek());
    falseCode.here(); }
}

```

- B12. At present, comparisons of a value with zero are easily achieved with code like

```

A @    ( push value of variable A )
0      ( push zero )
==     ( compare )

```

Show how, without modifying the opcode set of the PVM, one could introduce a new word to the dictionary

0==

which would allow a programmer to write, instead [10]

```
A @ 0==
```

One way is simply to add the word to the dictionary as one that emits three words of code - LDC 0 CEQ. Looking ahead to the possibility that there might be other three byte words we might write

```
insert("0==" , Word.THREWORD, PVM.ldc, 0, PVM.ceq);
```

with an addition to the Dictionary.insert method:

```
public static Word insert(String name, int kind, params int[] args) {
// Inserts word (with possible address fields) into dictionary
Word word = new Word(name, kind);
word.opcode = args[0];
if (kind == Word.TWOWORD)
word.address = args[1];
if (kind == Word.THREWORD) {
word.address = args[1];
word.address2 = args[2];
}
wordList.add(word);
return word;
}
```

and a tweak to the code generator:

```
switch (kind) {
case ONEWORD :
CodeGen.emit(opcode); break;
case TWOWORD :
CodeGen.emit(opcode); CodeGen.emit(address); break;
case THREWORD :
CodeGen.emit(opcode); CodeGen.emit(address); CodeGen.emit(address2); break;
}
```

In fact, for this exercise (only) we can do a simpler hack:

```
insert("0==" , Word.THREWORD);
```

and a tweak to the code generator:

```
switch (kind) {
case ONEWORD :
CodeGen.emit(opcode); break;
case TWOWORD :
CodeGen.emit(opcode); CodeGen.emit(address); break;
case THREWORD :
CodeGen.emit(PVM.ldc); CodeGen.emit(0); CodeGen.emit(PVM.ceq); break;
}
```

But the prize this year for the "I wish I had thought of that trick myself" competition goes to Rodain, who pointed out that all that was needed was to include the code

```
insert("0==" , Word.ONEWORD, PVM.not);
```

As I have often observed in the past, it is always exciting when a student teaches me something! Well done.

- B13. The system as supplied to you is dangerous, in that no checks have been introduced to ensure that control structures like IF ... ENDIF and REPEAT ... UNTIL are correctly balanced. Consider, for example, what would happen were the assembler to be presented with code like

```
UNTIL C @ 0 == REPEAT C ?
```

Suggest how such errors could be detected and reported. [10]

There are at least two strategies for this. One way would be to treat these words as special immutable ones, and change the grammar accordingly. Another would be to mark the labels in some way and check that they are

pushed and popped onto the label stack correctly:

```
public static void CodeForRepeat() {
    labels.push(new Label(known, Label.REPEAT));
}

public static void CodeForUntil() {
    if (labels.size() > 0 && labels.peek().labelKind() == Label.REPEAT)
        branchFalse(labels.pop());
    else Parser.SemError("repeat .. until unbalanced?");
}

public static void CodeForIf() {
    labels.push(new Label(!known, Label.IF));    // false part
    branchFalse(labels.peek());
}

public static void CodeForElse() {
    if (labels.size() >= 1 && labels.peek().labelKind() == Label.IF) {
        Label falseCode = labels.pop();
        labels.push(new Label(!known, Label.IF));    // endif exit
        branch(labels.peek());
        falseCode.here(); }
    else Parser.SemError("if ... endif unbalanced?");
}

public static void CodeForEndIf() {
    if (labels.size() >= 1 && labels.peek().labelKind() == Label.IF) {
        Label exit = labels.pop();
        exit.here();
    }
    else Parser.SemError("if ... endif unbalanced?");
}
```

What most submissions suggested was a simple "count the ifs" and "count the repeats", on the lines of

```
public static void CodeForRepeat() {
    labels.push(new Label(known, Label.REPEAT));
    repeatLevel++;
}

public static void CodeForUntil() {
    branchFalse(labels.pop());
    repeatLevel--;
}
```

followed at the end by a check that repeat was zero, but this is very inadequate. If, for example, one had source code reading

```
UNTIL "something" REPEAT
```

one would try to pop a label that had never been pushed. One might try to improve on this with code like

```
public static void CodeForUntil() {
    if (repeatLevel == 0) parser.SemError("repeat ... until unbalanced");
    else { branchFalse(labels.pop()); repeatLevel--; }
}
```

but this sort of thing won't prevent mistakes like the following where a repeat loop contains an if, but the endif only appears after the until!

```
REPEAT true IF a ? 0== UNTIL ELSE "lol" ENDIF
```

- B14. A very powerful feature in Forth is the ability for the user to define new words of his or her own choice. This is achieved by using so called "colon definitions". A construction like

```
: EVALUATE 0 >= IF "positive" pos ++ ELSE "negative" neg ++ ENDIF ;
```

defines the word EVALUATE to have the effect of examining the value at the top of the stack, displaying an appropriate message, and then incrementing a relevant counter. This word could then be used in code like

A @ EVALUATE B @ EVALUATE

with the implication that each time EVALUATE is encountered, the corresponding sequence of words is introduced into the code. This "macro" facility uses a colon-semicolon pair to demarcate this sequence, and the name of the word so defined is supplied immediately after the colon. As a second example, the following defines a word that will allow one to examine and print the value at the top of the stack as both an integer and character, but without discarding it:

```
: PEEK DUP . DUP .C ;
```

New words need not be limited to having alphanumeric names like EVALUATE and PEEK. One could introduce, for example

```
: 0== 0 == ;
```

as an alternative method for handling the situation described in B12.

Show how the colon-definition facility could be added to the assembler. Hint: This can be done elegantly in a relatively small number of lines of code. [20]

This is much easier than it might at first appear. We alter the grammar:

```
Directive      ( . string str;
                int value;
                Word word; .)
= "variable" Ident<out str> ( . Dictionary.AddVariable(str); .)
| "constant" Ident<out str>
  Number<out value>      ( . Dictionary.AddConstant(str, value); .)
| ":" Opcode<out str>    ( . ArrayList<Word> list = new ArrayList<Word>(); .)
  { OneWord<out word>    ( . if (word != null) list.add(word); .)
    } ";"               ( . Dictionary.addMacro(str, list); .)
.
```

Two new methods are added to the Dictionary class

```
public static Word insert(String name, int kind, ArrayList<Word> list) {
    // Inserts composite word or macro into dictionary
    Word word = new Word(name, kind);
    word.list = list;
    wordList.add(word);
    return word;
}

public static void addMacro(string name, ArrayList<Word> list) {
    // Adds a macro definition into dictionary
    insert(name, Word.MACRO, list);
}
```

And some additions are made to the Word class

```
public const int      // kinds of words
...
MACRO    = 11;

public String name;    // spelling
public int kind;       // one of the above kinds
public int opcode = 0; // basic opcode
public int address = 0; // offset for two-word opcodes
public ArrayList<Word> list = null; // list of component words (macros)

public void Compile() {
    // Each word is capable of assembling itself - very simple mapping
    switch (kind) {
        ...
        case MACRO :
            for (int i = 0; i < list.size(); i++)
                list[i].compile();
            break;
    }
```

Section C

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

Candidates were provided with the basic ideas of a Forth-like assembly language, and were invited to create an assembler that would generate PVM code from such source.

It was suggested that they might limit themselves to a system that would do the following:

- (a) handle strings, integer literals and character literals as code generators
- (b) Declare named variables and constants
- (c) Map simple "words" to the obviously equivalent PVM opcodes
- (d) Handle IF ... THEN and REPEAT ... UNTIL control constructs

They were provided with an exam kit for Java or C#, containing a working Parva compiler like that which they had used in the practical course, and with skeleton files for creating a tailored assembler, including a rudimentary code generator and dictionary handler. They were also given a suite of simple, suggestive test programs. 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 Assembler 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.