RHODES UNIVERSITY

Computer Science 301 - 2001 - Programming Language Translation

Well, here you are. Here is the free information you have all been waiting for, with some extra bits of advice:

- Don't panic. It is probably easier than you might at first think.
- The solution to section B does not need an enormous amount of code. But it needs rather careful thought. I am looking for evidence of mature solutions, not crude hacks.
- Do make sure you get a good night's sleep!

How to spend a Special Sunday

From now until two hours before the examination tomorrow, Computer Science 3 students have exclusive use of the Braae Laboratory. You are encouraged to throw out anyone else who tries to use it, but we hope that this does not need to happen. On Sunday evening Jody and I will have to move some computers around in preparation for Monday, but you will be able to continue working.

During this time you will be able to find the following files in the usual places, or by following links from the course web page:

- All the files as were made available for the practical course.
- The file `EXAM.ZIP` which contains the C++ version of the Coco/R system, and the sources for generating an "extended" CS301-2 compiler, essentially as in the model solution for Prac 21 this year, but with a few minor improvements. If you unzip this file you will be able to run GenMake, QEdit and Coco/R using your Favourite Language in the way that is, hopefully, familiar by now.
- Borland and Visual C++ compilers, which you can invoke in the usual way. You will not need to log on to a server to use these compilers; they are on the local C: drive.
- The entire set of source files for The Book, in a directory system starting at `I:CSC301\TRANS\SOURCES`, unpacked as described in Appendix A. The files are arranged by chapter and by language - for example, the C++ versions of the sources for chapter 18 are in `I:CSC301\TRANS\SOURCES\CHAP18\CPP`. If you have trouble locating these files you are free to ask for help (but the information is all in the file `I:CSC301\TRANS\SOURCES\README.SRC`).

Once you have copied the "exam kit" it is suggested that you reboot the machines into the "local" mode that you will use tomorrow. To do this you shut down, and then choose the "NONET" configuration as the system reboots. When the login screen reappears you log in with the username `test1` *and the password* `rbtest1`. *This works on all the machines; in this mode none of them have access to one another or to the network.*

Today you may use the files and the systems in any way that you wish subject to the following restrictions: *Please observe these in the interests of everyone else in the class.*

(a) Create a working directory on D: in the usual way, and preferably do all your work within this directory.

(b) When you have finished working, **please** delete your files from the D: drive, so that others are not tempted to come and snoop around to steal ideas from you.

(c) Since tomorrow you will not have access to the file server, work on the D: drive and not on your server file space, for practice, if for no other reason!

(d) You are encouraged to discuss the problem with one another, and with anybody not on the "prohibited" list.

(e) You are also free to consult books in the library. If you cannot find a book that you are looking for, it may well be the case that there is a copy in the Department. Feel free to ask.

I suggest that you DO spend some of the next 24 hours in DISCUSSION with one another, and some of the time in actually TRYING OUT your ideas. You have plenty of time in which to prepare and test really good solutions

- go for it and good luck! Remember that you may not bring any papers or diskettes into the exam room tomorrow.

If you cannot unpack the file, or have trouble getting the familiar tools to work (unlikely!), you may ask me for help. You may also ask for explanation of any points in the question that you do not understand, in the same way that you are allowed to ask before the start of an ordinary examination. You are no longer allowed to ask me questions about any other part of the course. Sorry; you had your chance earlier, and I cannot allow this without risking the chance of sneak questions and suggestions being called for.

If you cannot solve the problem completely, don't panic. It has been designed so that I can recognize that students have reached varying degrees of sophistication and understanding.

How you will spend a Merry Monday

Just before the start of the formal examinations the laboratory will be unavailable. During that time

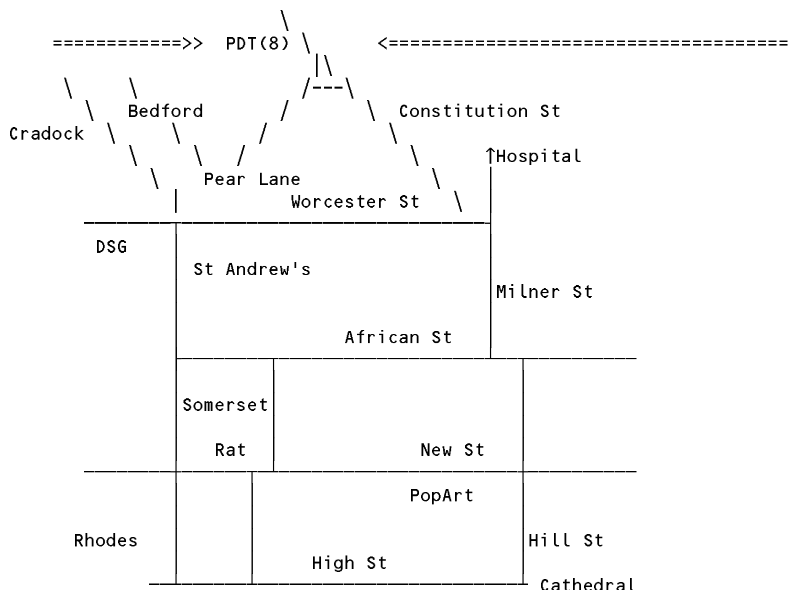
- The `D:` and `C:\TEMP` drives will be cleared.
- The network connections will be disabled.

At the start of each examination session:

- You will receive ordinary answer books for the preparation of answers, and an examination paper.
- You will be supplied with a diskette on which you will find the exam kit. This will contain the Coco/R system, as was made available on the previous day, and also various machine readable parts of the examination paper itself, in files with names like `Q7.TXT` (Question 7).
- A few copies of Chapter 12 of the text ("Using Coco/R - Overview") will be available as free information, should you need them.
- At the end of the exam you will be given a chance to copy any files that you have edited or created back to the diskette. It is, obviously, in your own interest to make sure that you know how to copy files correctly.

Cessation of Hostilities Party

As previously mentioned, Sally and I would like to invite you to an informal end-of-course party at our house on 12 November. There's a map below to help you find your way there. It would help if you could let me know whether you are coming so that we can borrow enough glasses, plates etc. E-mail to p.terry@ru.ac.za. Time: from 18h30 onwards. Dress: Casual



And for my last trick -

Section B [85 marks]

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

For several years your predecessors in this course - and even yourselves - have been expected, as part of the practical course, to gain an understanding of a stack machine architecture by preparing programs written in a very limited form of assembler language in which all addressing had to be done in terms of numerical values which students were supposed to calculate for themselves - with understandable frustration setting in every time they inserted or deleted a few statements into programs as they debugged them. During this time students have begged me to give them a "real" assembler in which alphanumeric labels could be used to identify constants, variables, and the destinations of branch instructions. I have, of course, always been too busy to do this, but with 24 hours at your disposal and the expert knowledge you have amassed after studying the translators course this year, you should be able to remedy this situation - and if you succeed you may be able to make some useful pocket money selling your system to the class next year!

We start by observing that, rather than writing code as exemplified by the columns on the left, most people would prefer to write code as exemplified by the columns on the right

```
ASSEM  $D+          |  ASSEM   $D+                   # Read a list and write it backwards
                    |    CONST                       # High level declarations
                    |      Max = 10;                 # constants
                    |      Width = 6;
                    |    INT                         # variables
BEGIN               |      List[Max], I, Item;
  DSP    13         |    BEGIN                       # DSP 13 can be generated automatically
  ADR   -12         |             ADR     I          # I := 0;
  LIT     0         |             LIT     0          #
  STO               |             STO                #
  ADR   -13         |       READ  ADR     Item       # LOOP
  INN               |             INN                #   Read(Item);
  ADR   -13         |             ADR     Item       #
  VAL               |             VAL                #
  BZE    32         |             BZE     DONE       #   IF Item = 0 THEN EXIT END;
  ADR    -1         |             ADR     List       #
  ADR   -12         |             ADR     I          #
  VAL               |             VAL                #
  LIT    11         |             LIT     SIZE(List) #
  IND               |             IND                #
  ADR   -13         |             ADR     Item       #
  VAL               |             VAL                #
  STO               |             STO                #   List[I] := Item;
  ADR   -12         |             ADR     I          #   I++;
  PPP               |             PPP                #
  BRN     7         |             BRN     READ       # END;
  PRS   'Reversed'  |       DONE  PRS     'Reversed' # Write('Reversed');
  ADR   -12         |       PRINT ADR     I          # WHILE I > 0 DO
  VAL               |             VAL                #
  LIT     0         |             LIT     0          #
  GTR               |             GTR                #
  BZE    59         |             BZE     EXIT       #
  ADR   -12         |             ADR     I          #   I--;
  MMM               |             MMM                #
  ADR    -1         |             ADR     List       #
  ADR   -12         |             ADR     I          #
  VAL               |             VAL                #
  LIT    11         |             LIT     Max + 1    #
  IND               |             IND                #
  VAL               |             VAL                #
  LIT     6         |             LIT     Width      #   Write(List[I] : 6);
  PRN               |             PRN                #
  BRN    34         |             BRN     PRINT      # END
  HLT               |       EXIT  HLT                # RETURN
END.                |    END.
```

Since it might be dangerous to place too much reliance on what would be required of an assembler, or indeed determine exactly what is permitted in the assembler language itself from studying this one single example, here are some suggestions for deriving a complete system. (In the exam kit will be found some other example

programs to assist in your development of the assembler.)

(a) You can make use of Coco/R, and in particular derive a solution by making use of the attributed grammar and support modules (symbol table handler, code generator, error handler, frame files etc) that were useful in the development of a CS301 compiler/interpreter.

(b) The assembler statements should appear between a bracketing `BEGIN` and `END`, and may optionally be preceded by declarations of constants and variables (like `Max`, `Width`, `List`, `I` and `Item`) using similar syntax to that found in CS301 programs.

(c) The assembler system should be able to assemble simple programs in which the addressing is all given in absolute form (as in the example on the left), as well as those with alphanumeric names and labels.

(d) Treat the mnemonics as key (reserved) words. Since `AND` and `NOT` are mnemonic opcodes, use `!`, `&&` and `||` for Boolean operators.

(e) Alphanumeric labels (like `READ`, `PRINT`, `DONE` and `EXIT`) used as the targets of branch instructions must be uniquely defined. For simplicity, these labels should not be allowed to duplicate identifiers used in the declaration of named constants or variables.

(f) It is acceptable to define labels without ever having branch instructions that referred to them, to have multiple labels defined at one point, or to have multiple branches to one point, for example

```
        BRN   START            # Unnecessary, but legal
START
LOOP    LIT   6
        LIT   7
        PRN
        BRN   START            # equivalent to BRN LOOP
```

(g) It would not be acceptable to have branch instructions refer to labels that are never defined, for example

```
BEGIN
  LOOP    LIT   6
          LIT   7
          PRN
          BRN   START            # Start is undefined
END
```

(h) The `LIT` and `DSP` mnemonics should be allowed to take a constant-generating expression as a parameter:

```
DSP  6                  # Absolute form
LIT  Max                # Equivalent to LIT  10
LIT  Max * 10 + Width   # Equivalent to LIT  106
LIT  Size(Array)        # Equivalent to LIT  11
```

where `Size` is a pseudo function that can return the storage space needed for the variable quoted as its actual argument (this would clearly be useful in applications that use arrays in particular).

(i) The `ADR` mnemonic should be allowed to take a (possibly signed) number or a variable name as its parameter. In the case where this name refers to an array a possible extension would be to allow it to have a constant subscript indicating a further offset that could be computed at assemble time, for example:

```
ADR  -1                 # absolute addressing
ADR  Item               # equivalent to ADR -13
ADR  List               # equivalent to ADR -1
ADR  List[0]            # equivalent to ADR -1
ADR  List[2]            # equivalent to ADR -3
```

(j) Not much attention need be paid to type checking - at this level programmers should be relied on to get these semantics correct for themselves.

(k) Apart from situations where they are necessary for separating other alphanumeric quantities, whitespace characters may be used at the coder's discretion to improve the appearance of source code.

(l) In the extended compiler for CS301 you may have made use of additional opcodes to the ones listed below,

in particular to handle switch/case statements. For the purposes of this examination you may confine your assembler to the opcodes in the table on page 7.

Instruction set for stack machine

Several of these operations belong to a category known as **zero address** instructions. Even though operands are clearly needed for operations such as addition and multiplication, the addresses of these are not specified by part of the instruction, but are implicitly derived from the value of the stack pointer SP. The two operands are assumed to reside on the top of the stack and just below the top; in our informal descriptions their values are denoted by TOS (for "top of stack") and SOS (for "second on stack"). A binary operation is performed by popping its two operands from the stack into (inaccessible) internal registers in the CPU, performing the operation, and then pushing the result back onto the stack.

```
AND      Pop TOS and SOS, and SOS with TOS, push result to form new TOS
ORR      Pop TOS and SOS, or  SOS with TOS, push result to form new TOS
ADD      Pop TOS and SOS, add SOS to TOS, push sum to form new TOS
SUB      Pop TOS and SOS, subtract TOS from SOS, push difference to form new TOS
MUL      Pop TOS and SOS, multiply SOS by TOS, push product to form new TOS
DVD      Pop TOS and SOS, divide SOS by TOS, push quotient to form new TOS
REM      Pop TOS and SOS, divide SOS by TOS, push remainder to form new TOS
EQL      Pop TOS and SOS, push 1 to form new TOS if SOS = TOS, 0 otherwise
NEQ      Pop TOS and SOS, push 1 to form new TOS if SOS ≠ TOS, 0 otherwise
GTR      Pop TOS and SOS, push 1 to form new TOS if SOS > TOS, 0 otherwise
LSS      Pop TOS and SOS, push 1 to form new TOS if SOS < TOS, 0 otherwise
LEQ      Pop TOS and SOS, push 1 to form new TOS if SOS ≤ TOS, 0 otherwise
GEQ      Pop TOS and SOS, push 1 to form new TOS if SOS ≥ TOS, 0 otherwise
NEG      Integer negation of TOS
NOT      Boolean negation of TOS

STK      Dump stack to output (useful for debugging)
PRN      Pop TOS and SOS, write SOS to output as an integer value in field width TOS
PRB      Pop TOS and SOS, write SOS to output as a Boolean value in field width TOS
PRS  A   Write the nul-terminated string that is assumed to be stacked in the literal pool from Mem[A]
NLN      Write a newline (carriage-return-line-feed) sequence
INN      Read integer value, pop TOS, store the value that was read in Mem[TOS]
INB      Read Boolean value, pop TOS, store the value that was read in Mem[TOS]

DSP  A   Decrement value of stack pointer SP by A
LIT  A   Push the integer value A onto the stack to form new TOS
ADR  A   Push the value BP + A onto the stack to form new TOS. (This value is conceptually the address
         of a variable stored at an offset A within the stack frame pointed to by the base register BP.)
IND      (Range checked indexing of array) Pop TOS to yield Size; pop TOS and SOS; if 0 ≤ TOS < Size
         then subtract TOS from SOS, push result to form new TOS
INX      (Unchecked indexing of array) Pop TOS and SOS, subtract TOS from SOS, push result to form new TOS
VAL      (Dereferencing) Pop TOS, and push the value of Mem[TOS] to form new TOS
DUP      Push TOS to form duplicate copy
STO      Pop TOS and SOS; store TOS in Mem[SOS]
PPP      Pop TOS and increment Mem[TOS] by 1
MMM      Pop TOS and decrement Mem[TOS] by 1

HLT      Halt
BRN  A   Unconditional branch to instruction A
BZE  A   Pop TOS, and branch to instruction A if TOS is zero
BAN  A   Branch to instruction A if TOS is false; else pop TOS
BOR  A   Branch to instruction A if TOS is true; else pop TOS
NOP      No operation
```

The instructions in the first group are concerned with arithmetic and logical operations, those in the second group afford I/O facilities, those in the third group allow for the access of data in memory by means of manipulating addresses and the stack, and those in the last group allow for control of flow of the program itself.

Examination period allocations

The morning session will run from 08h30 until 11h30. Candidates must be in the Struben Building from 08h15, and will not be allowed to leave until everyone has finished and the papers been collected.

The afternoon session will run from about 12h00 until 15h00. Candidates must assemble in the Hamilton Building from 11h15, as we have to make sure that there is no collaboration between sessions.

Session	Student no.	Name
afternoon	699A0124	Abrey, DM
afternoon	697A1002	Adar, AO
morning	699A0269	Alli, S
morning	699A0928	Amtha, P
afternoon	699A0564	Anderson, ML
afternoon	699A1450	Armstrong, MEG
morning	698A1083	Asafo-Adjei, T
morning	69750204	Bhe, U
morning	697B3411	Brown, DE
afternoon	699C1815	Cave, ML
morning	699D2774	Dada, A
morning	699E1291	Everett, KA
morning	699K1690	Eyambe, LK
morning	699F0023	Ferguson, BA
afternoon	698F4230	Futshane, Z
afternoon	699G1912	Gaul, T
morning	699G0578	Goodenough, TW
morning	699H3447	Haig, D
afternoon	699H1589	Hatting, CA
morning	698H1625	Hone, GJ
morning	699J1570	Jogee, ZS
afternoon	698J4312	Jones, EB
morning	699K2455	Keildson, J
morning	699K0958	Kieser, SJ
morning	699K3906	Krijger, HGL
afternoon	697K5261	Ku, YJ
afternoon	698L3455	Leong Son, JMJ
afternoon	699M1325	Mabvudza, FTL
afternoon	699M1260	Machimana, TD
afternoon	699M0302	Mackie, DS
afternoon	69731436	Mahluza, NLN
afternoon	698M6232	Manners, PJ
morning	699M0189	Manyindo, SN
morning	699M1972	Mathebula, DW
morning	698M1813	Matlock, JA
morning	698M1871	Miya, SC
morning	697M6120	Mkuzangwe, NNP
morning	699M2744	Moeca, WT
morning	699M0761	Morkel, C
morning	699M0158	Mpofu, AC
afternoon	699N3374	Nadasen, KG
afternoon	698N3985	Naidoo, D
morning	699N0776	Naidoo, O
afternoon	699N0640	Naidoo, P
afternoon	699N0225	Nel, LG
afternoon	699N0587	Nyamadzawo, F
morning	699O2038	Okai-Tettey, HA
afternoon	698P3127	Parbhoo, NH
afternoon	699P1968	Parbhu, P
morning	699P1628	Pare, JR
morning	699P3117	Proske, H
afternoon	699R2070	Rampertab, M
afternoon	698R1293	Rana, B
morning	698R3067	Renwick, MR
morning	699R0191	Rudraraju, U
morning	699S2055	Seedat, RG
afternoon	699S2711	Shepherd, J
morning	699S1482	Sigsworth, R
afternoon	600S1452	Smit, NK
morning	699S0115	Sterne, PJ
afternoon	600T4417	Thomas, JJ
afternoon	699U1402	Ubogu, CO
afternoon	699V1803	Van Berkel, GG
afternoon	699V0697	Vanda, NM
afternoon	699W0746	Wasswa, PJK
afternoon	699Y1212	Yazdani, N
morning	699Y1855	Yong, RAH

Other Goodies

The exam kits contain a selection of silly assembler programs which you may find useful in testing the system you develop. They are not all "correct" of course.

You will also find an executable `ASM.EXE` derived from my model solution to this exercise. Don't bother to try anything like reverse engineering this; it has all been encrypted!

A command like

```
ASM A02.ASM
```

will attempt to assemble the code in the file `A02.ASM` and then interpret it.

There are two versions of the attributed grammar for the CS301 compiler. `CS301.ATG` uses two "expression parsers" as suggested in the prac course - one for constant expressions and the other for "code generating" expressions. `CS301A.ATG` is a version as discussed in the prac solution where only one, suitably parameterized, expression parser is used.

Listings of `CS301.ATG`, of the code generator `CGEN.CPP` and of the table handler and interpreter interfaces `TABLE.H` and `STKMC.H` are available today, and will be made available tomorrow (when the printers are disabled).

Special attention is drawn to some features of this system which have changed slightly since the last practical solution was issued, or which you may have overlooked at the time:

- Take note of the useful routine `STKMC::opcodes` .
- Take note of the routine `TABLE::update` which has been added to the symbol table handler.
- Take note of the `LexName` routine, which is an alternative to `LexString` useful in situations where one is ignoring case sensitivity in source code.
- In the original CS301 language the keywords `NOT`, `AND` and `OR` were used in formulating Boolean expressions. In the version you now have, the C++-like alternatives `!`, `&&` and `||` are also allowed. Since the keywords will clash with stack-machine opcodes, in your assembler limit yourself to the C++-like versions of these operators. The alternatives are clearly delineated in the attribute grammars you have been given.

  I apologise - there is a typo in the listing of CS301.ATG. Line 445 should read

  ```
  | NotOp CUnaryExp<u, value> (. if (!booltypes.memb(u)) SemError(218);
  ```

  The correct line is to be found in the file as supplied to you

And here are two hints:

- Don't be over ambitious at the outset. You could perhaps begin by developing an "absolute assembler"which might be capable of handling examples like `A01.ASM, A02.ASM, A03.ASM, A05.ASM`. When you have that working, go on to incorporate the variables and expressions. Leave the "labels" as exemplified from about `A14.ASM` till later.
- It is suggested you use the `$D+/$D-` pragmas in your assembler system in a similar way to that used in the compiler to help you debug your system.