RHODES UNIVERSITY

November Examinations - 2001

Computer Science 301 - Paper 2

Examiners:
Prof P.D. Terry
Prof E.H. Blake

Time 3 hours
Marks 180
Pages 7 (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 the full text of Section B. During the examination, candidates were given machine executable versions of the Coco/R compiler generator, access to a computer and machine readable copies of the questions.)

Section A [95 marks]

1. Draw a diagram clearly depicting the various phases found in a typical compiler. Indicate which phases belong to the "front end" and which to the "back end" of the compiler. [9 marks]

2. (a) What is meant by the term "self-compiling compiler"? [2 marks]

   (b) Describe (with the aid of T-diagrams) how you would perform a "half bootstrap" of a compiler for language X, given that you have access to the source and object versions of a compiler for X that can be executed on machine OLD and wish to produce a self-compiling compiler for language X that can be executed on machine NEW. [10 marks]

3. Consider the following grammar expressed in EBNF for describing the progress of a typical university course:

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

   (a) Develop a recursive descent parser for the grammar (matching the EBNF as given above). Assume that you have suitable `Abort`, `Accept` and `GetSym` routines (which you need not develop), and that `GetSym` decodes `Sym` as one of the tokens below. Your system should detect errors, of course, but need not incorporate "error recovery". [12 marks]

```
{ EOFSym, lectSym, hanSome, pracSym, testSym, tutSym, PANICSYM, examSym, unknownSym }
```

   (b) What do you understand by the statement "two grammars are equivalent"? [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 [...]. [5 marks]

   (d) Analyse the equivalent grammar derived in (c) to determine whether it obeys the LL(1) constraints. [8 marks]

   (e) If you found that the grammar did not obey the LL(1) constraints, does that mean that the parser produced in (a) would fail for some valid inputs? If so, give an example of input that could not be parsed; otherwise justify your claim that it would always succeed. [3 marks]

4. The CS301 language for which a compiler was developed in this course allows for various statements, including a "while" loop. Relevant parts of the attributed grammar are shown below.

```
StatementSequence
=  Statement { WEAK ";" Statement } .

Statement
=  SYNC [   Assignment
          | IfStatement       | WhileStatement
          | ReadStatement     | WriteStatement
          | ReturnStatement   | CaseStatement
        ] .

WhileStatement
=                             (. CGEN_labels startloop, testlabel, dummylabel; .)
   "WHILE"                    (. CGen->storelabel(startloop); .)
   Condition "DO"             (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   StatementSequence          (. CGen->jump(dummylabel, startloop);
                                 CGen->backpatch(testlabel); .)
   "END" .
```

An enthusiastic language extender has suggested that CS301 would be greatly improved by the addition of a post-test loop, and has come up with two possibilities:

```
PostTestStatement = "DO" StatementSequence "WHILE" Condition .
```

or

```
PostTestStatement = "DO" StatementSequence "UNTIL" Condition .
```

(a) Advise her, with reasons, as to whether or not either or both of these suggestions would be acceptable, and which (if either) would be preferable. [5 marks]

(b) For the form of your choice, show how the grammar above would be extended to recognise the statement form and generate correct code. [5 marks]

5. As you should recall, CS301 is a "strictly typed" language, and expressions of the form

```
NOT 6
```

or

```
TRUE > 56
```

or

```
3 + 4 AND 5
```

are unacceptable. Most strictly typed languages allow for programmers to circumvent (bypass) these restrictions, typically by allowing so-called "type casting", as exemplified by

```
NOT BOOL(6)
```

or

```
INT(TRUE) > 56
```

or

```
3 + INT(BOOL(4) AND BOOL(5))
```

(a) In the free information for this paper appears an attributed grammar for generating code to evaluate expressions which does not incorporate such type casting. Show how the grammar could be altered to do so. (Only write out those parts that would have to change.) [6 marks]

(b) If strict type checking can be bypassed in this way, what advantages or disadvantages do strictly typed languages possess over languages like C++, where Boolean, integer and character types are all compatible? [4 marks]

6. As you should be aware, IP addresses as used in Internet communication are typically expressed in "dotted decimal" form, as exemplified by `146.231.128.6`. The IP address actually represents a 32 bit integer; the four numbers in the quadruple corresponding to successive 8 bit components of this integer. For humans, machine addressing is usually more memorable when expressed in "DNS" format, as exemplified by `terrapin.ru.ac.za`. Some systems maintain tables of matching addresses, for example

```
146.231.122.13      cspt1.ict.ru.ac.za    #comments appear like this
146.231.128.6       terrapin.ru.ac.za
146.231.56.10       thistle-sp.ru.ac.za
147.28.0.62         psg.com
```

When we moved our CS and IS departments to new premises recently, a decision was made to rename and uniquely renumber all the many machines in our possession. Our system administrators tried to draw up a table like the one above, which was then merged with the existing table in the IT division. Unfortunately, a few mistakes were made, which caused havoc until they were ironed out. For example, there were lines reading

```
146.231.122.11235   cspt1.ict.ru.ac.za    #invalid IP address
146.231.122.15      cspt2.ict.ru.ac.za
146.231.122.15      cspt3.ict.ru.ac.za    # non-unique IP address
```

Complete the ATG file below to show how Coco/R could be used to develop a system that would enable a file in this format quickly to be checked and the errors identified. (Hint: make use of the template list handling class that proved useful in various other applications in this course, the interface to which is provided below). There is no need to reproduce the code below; it will suffice merely to develop the TOKENS and PRODUCTIONS section in your solution, and to indicate what, if any declarations and/or `#include` lines would be added at the beginning. [24 marks]

```
COMPILER CheckIP $XCN

IGNORE CASE
IGNORE CHR(1) .. CHR(31)

CHARACTERS
  digit   = "0123456789" .
  letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  eol     = CHR(10) .

COMMENTS FROM "#" TO eol

TOKENS

PRODUCTIONS


END CheckIP.
```

Generic List Template Class

```
/* Simple generic linked list class.  Only the interface is shown here
   George Wells  --  27 February 1996 */

template<class T> class List
  { public:
      List (void);                               // Constructor
      List (const List& lst);                    // Copy constructor
      ~List (void);                              // List destructor
      void add (T item, int position = INT_MAX); // Place new item in a List
      void remove (int position, T &item);       // Remove item at position in List
      int length (void);                         // Return number of elements in List
      T& operator[] (int index);                 // Subscript a List
      int position (T item);                     // Return position item in a List (or -1)
      int isMember (T item);                     // True if item is in List
  }; // class List
```

Expression parser from CS 301 compiler - Question 5

```
Expression<TABLE_types &e>
=                              (. TABLE_types a;
                                  CGEN_labels shortcircuit; .)
   AndExp<e>
   { "OR"                      (. if (shortBoolean) CGen->booleanop(shortcircuit, CGEN_opor) .)
     AndExp<a>                 (. if (!(booltypes.memb(e) && booltypes.memb(a)))
                                    { SemError(218); e = TABLE_none; }
                                  else e = TABLE_bools;
                                  if (shortBoolean) CGen->backpatch(shortcircuit);
                                  else CGen->binarybooleanop(CGEN_orrop); .)
   } .


AndExp<TABLE_types &a>
=                              (. TABLE_types e;
                                  CGEN_labels shortcircuit; .)
   RelExp<a>
   { "AND"                     (. if (shortBoolean) CGen->booleanop(shortcircuit, CGEN_opand) .)
     RelExp<e>                 (. if (!(booltypes.memb(a) && booltypes.memb(e)))
                                    { SemError(218); a = TABLE_none; }
                                  else a = TABLE_bools;
                                  if (shortBoolean) CGen->backpatch(shortcircuit);
                                  else CGen->binarybooleanop(CGEN_andop); .)
   } .


RelExp<TABLE_types &r>
=                              (. TABLE_types a;
                                  CGEN_operators op; .)
   AddExp<r>
   [ RelOp<op> AddExp<a>       (. if (r == TABLE_bools || a == TABLE_bools) SemError(218);
                                  r = TABLE_bools; CGen->comparison(op) .)
   ] .

AddExp<TABLE_types &a>
=                              (. TABLE_types m;
                                  CGEN_operators op; .)
   MultExp<a>
   { AddOp<op> MultExp<m>      (. if (!(arithtypes.memb(a) && arithtypes.memb(m)))
                                    { SemError(218); a = TABLE_none; }
                                  else CGen ->binaryintegerop(op); .)
   } .

MultExp<TABLE_types &m>
=                              (. TABLE_types u;
                                  CGEN_operators op; .)
   UnaryExp<m>
   { MulOp<op> UnaryExp<u>     (. if (!(arithtypes.memb(m) && arithtypes.memb(u)))
                                    { SemError(218); m = TABLE_none; }
                                  else CGen ->binaryintegerop(op); .)
   } .

UnaryExp<TABLE_types &u>
=    Factor<u>
   | "+" UnaryExp<u>           (. if (!arithtypes.memb(u)) {
                                    SemError(218); u = TABLE_none; } .)
   | "-" UnaryExp<u>           (. if (!arithtypes.memb(u)) {
                                    SemError(218); u = TABLE_none; }
                                  else CGen->negateinteger(); .)
   | "NOT" UnaryExp<u>         (. if (!booltypes.memb(u)) SemError(218);
                                  else CGen->negateboolean();
                                  u = TABLE_bools; .) .

Factor<TABLE_types &f>
=                              (. int value;
                                  TABLE_entries entry; .)
     Designator<classset(TABLE_consts, TABLE_vars), entry>
                               (. f = entry.type;
                                  switch (entry.idclass)
                                  { case TABLE_vars :
                                      CGen->dereference(); break;
                                    case TABLE_consts :
                                      CGen->stackconstant(entry.c.value); break;
                                  } .)
   | Number<value>             (. CGen->stackconstant(value); f = TABLE_ints; .)
   | "TRUE"                    (. CGen->stackconstant(1); f = TABLE_bools .)
   | "FALSE"                   (. CGen->stackconstant(0); f = TABLE_bools .)
   | "(" Expression<f> ")" .
```

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+



BEGIN
  DSP    13
  ADR   -12
  LIT     0
  STO
  ADR   -13
  INN
  ADR   -13
  VAL
  BZE    32
  ADR    -1
  ADR   -12
  VAL
  LIT    11
  IND
  ADR   -13
  VAL
  STO
  ADR   -12
  PPP
  BRN     7
  PRS   'Reversed'
  ADR   -12
  VAL
  LIT     0
  GTR
  BZE    59
  ADR   -12
  MMM
  ADR    -1
  ADR   -12
  VAL
  LIT    11
  IND
  VAL
  LIT     6
  PRN
  BRN    34
  HLT
END.
```

```
ASSEM   $D+                        # Read a list and write it backwards
  CONST                            # High level declarations
    Max = 10;                      # constants
    Width = 6;
  INT                              # variables
    List[Max], I, Item;
  BEGIN                            # DSP 13 can be generated automatically
           ADR     I               # I := 0;
           LIT     0               #
           STO                     #
     READ  ADR     Item            # LOOP
           INN                     #   Read(Item);
           ADR     Item            #
           VAL                     #
           BZE     DONE            #   IF Item = 0 THEN EXIT END;
           ADR     List            #
           ADR     I               #
           VAL                     #
           LIT     SIZE(List)      #
           IND                     #
           ADR     Item            #
           VAL                     #
           STO                     #   List[I] := Item;
           ADR     I               #   I++;
           PPP                     #
           BRN     READ            # END;
     DONE  PRS     'Reversed'      # Write('Reversed');
     PRINT ADR     I               # WHILE I > 0 DO
           VAL                     #
           LIT     0               #
           GTR                     #
           BZE     EXIT            #
           ADR     I               #   I--;
           MMM                     #
           ADR     List            #
           ADR     I               #
           VAL                     #
           LIT     Max + 1         #
           IND                     #
           VAL                     #
           LIT     Width           #   Write(List[I] : 6);
           PRN                     #
           BRN     PRINT           # END
     EXIT  HLT                     # RETURN
  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` $\neq$ `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` $\leq$ `TOS`, 0 otherwise
`GEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` $\geq$ `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.

Computer Science 301 - November 2001 - Paper 2 - Answer sheet for Question 6.

Hand this page in with your answer book - fill in your student number clearly . . . .

```
COMPILER CheckIP $XCN

IGNORE CASE
IGNORE CHR(1) .. CHR(31)

CHARACTERS
  digit   = "0123456789" .
  letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  eol     = CHR(10) .

COMMENTS FROM "#" TO eol

TOKENS




PRODUCTIONS
















END CheckIP.
```