B1 Compilers (25 pts)
SYMBOL TABLE MANAGEMENT AND ERROR DETECTION: (Complete all parts.)

As the main data structure for storing and retrieving information about user-defined names in a program, the implementation of a symbol table typically supports the following operations:

- insertion of a new entry
- lookup an entry based on a name

(a) [8 pts] Assume that the language you are compiling is an imperative language that has global and local variable declarations, but function declarations cannot be nested. An example of such a language is C confined to a single file. Describe a data structure, how the basic operations above could be implemented, and when to call these routines to support symbol table management for this language.

(b) [10 pts] Assume that you need to build a compiler for an imperative language that has global and local variable declarations, but now functions can be nested. The static scoping rule used for accessing variable names is the commonly known most closely nested rule. Assume that names must be declared before they are used. An example of such a language is Pascal. Describe a data structure, how the basic operations above could be implemented, any additional operations that need to be implemented, and when to call these routines to support symbol table management for this language.

(c) [7 pts] ANSWER ONE of the following questions:

(i) Describe how a type checker could deal with each of the following: (1) coercion, and (2) overloaded symbols (e.g., * for multiply and pointers).

(ii) Describe the tradeoffs in using the following intermediate code representations: abstract syntax trees, concrete syntax trees, dags, and 3-address code.

(iii) Describe two errors that a compiler could find using a symbol table, that could not be found earlier in the compilation, and briefly explain how they could be detected.

B2 Compilers (25 pts)
RUN-TIME STORAGE MANAGEMENT AND CODE GENERATION: (Complete all parts.)

(a) [15 pts] Consider the following code segment, static scoping with nested functions, and a stack storage allocation scheme. Note that the keyword **ref** is used to indicate call by reference, while **val** is used to indicate call by value. **sub1** and **sub2** are nested inside **main**. **sub3** is nested inside **sub2**.

Show the organization of the activation record stack, including the order of activation records and for each activation record, the contents including the local variables, parameter information, control link, and access link at each numbered point **(num)** in the code. The order in which fields are arranged within an activation record and the actual value of each variable are not important. Just indicate the name if the actual value is stored; otherwise, indicate addr(name) if the address of that location is stored. Be sure to show the actual-formal parameter associations. At points (2) and (4), also show the contents of a display if you were using a display for accessing nonlocals.

```
main()
{
 int g1, g2;  /*globals*/

 void sub1(val int p1; ref int p2)
 {                                   **(1)**
   int s1,s2,s3;
    call readit(s1,s2,s3);
   call sub2(p2);
    p2 = p1 * s1 + s2 + s3;
 } /* end sub1*/

 void sub2(ref int p3)
 {
    int s4,s5;
    void sub3(val int p4,p5)
    {                                **(3) on first call,  **(4)** on third call
       int s6;
       s6 = p5;
        if p4 == 4 then call sub3(p4-1,p5);  /* evaluates to true only on second call*/
    } /*end sub3*/

  s4 = p3;                  **(2)** on first call only
   s5 = 5;
  call sub3(s4,s5);
  } /*end sub2*/

g1 = 3;
g2 = 4;
call sub1(g2,g1);
call sub2(g2);
} /*end main*/
```

question continued on next page...

B2 Compilers (25 pts) continued

(b) [10 pts] ANSWER ONE of the following:

(i) In register allocation, one decision that has to be made is what policy to use when there are no more empty registers and a register is needed to hold a variable to be used for a particular instruction. Describe a policy that could be used in this situation and discuss the motivation for the policy and when it might fail to perform a good allocation.

(ii) Some errors can not be detected until run-time, but a good compiler should generate code to detect and deal gracefully with the error rather than leaving it to the operating system to give the user a nondescriptive error message. Describe one such error, and describe how the code generator could generate code to detect and gracefully deal with the error.

(iii) Compiler optimizations can be applied interprocedurally, globally, or locally. For each of these classes, describe a common program representation used in the analysis and describe one example optimization performed in this class. You can use an example code segment to demonstrate the optimization if you like.

B3 Compilers (25 pts)

LEXICAL ANALYSIS AND PARSING: (Answer all questions.)

(a) [13 pts] Write an algorithm that takes as input a regular expression and generates a context-free grammar for the language described by that regular expression.

(b) [12 pts] How do the constructions of SLR, Canonical-LR and LALR tables differ from each other?

B4 Compilers (25 pts)

LANGUAGE IMPLEMENTATION: (Answer all questions.)

Broadly speaking, the trend from assembly languages to high-level languages has been to make programming languages more like human languages. Suppose you were asked to write a compiler with the features below. Discuss how the scanner, parser, symbol table manager, and code generator would have to be constructed to implement these features. What data structures and algorithms would have to be used? Can this compiler be implemented with the help of Lex and Yacc? Explain why or why not.

The desired features are:

(a) [13 pts] Conditional actions that are described by giving the general case first, followed by the exceptions. For example,

```
{ Let x.tax = .30 * x.income;
  Print "The normal tax is", x.tax;
}
But if (rich(x)) x.tax = .50 * x.income;
But if (graduatestudent(x))
  { x.tax = 0;
    Print "Must be a graduate student!";
  }
```

(b) [12 pts] Assume that normal static scoping rules apply. Now, instead of specifying the full name of an access to a field of a record variable, only the field name needs to be specified to unambiguously identify the field being accessed.

For example, rather than specifying **x.city** to access the field **city** of record variable **x**, we could specify just **city** as in:

```
if (city = = detroit) push x onto cold-climate-list;
```

when **x** is the only statically visible variable with the field **city** and there is no other visible variable called **city**.