Math 274
Final Exam

time : 2 1/2 hrs. Fall 1995-96

G. Jalloul

1. **(81%)** Consider the following SPL program

```
integer array (10) data;
integer sum, limit;

procedure Initialize ( integer array a);
   integer i;
      integer function GetLimit;
         read limit
      return limit;                                   (1)

   limit := GetLimit;
   i:=1;                                              (2)
   while i<= limit do
         read a(i);  i:= i+1;
   end;
return;                                               (3)

procedure Add( integer array a;  integer m; integer var s);
   integer j;

   j:=1; s:=0;
   while j<= m do
         s:= s+a(j);  j:= j+1;
      end;
return;                                               (4)

   Initialize(data);                                  (5)
   Add(data, limit, sum);                             (6)
   write sum;                                         (7)
end.
```

Input : 10 100 101 102 109

a) In each of the following cases, show by diagram the run-time stack. For each diagram give the value of the stack pointer and any display registers that may be needed. Also show clearly the contents of different stack locations and the addresses of these locations. Assume that the stack starts at location 4000 in memory and that it grows from low to high memory. Use the statement numbers given above as return addresses in case any is needed.
   1. The stack just before executing statement (5)
   2. The stack resulting from the call to procedure *Initialize* (statement 5) and just before the return from *Getlimit* in statement (1)
   3. The stack resulting from the call to procedure initialize (statement 5) and just before the return from this procedure in statement (3)
   4. The stack resulting from the execution of statements in the main program including the call to procedure add (statement (6) and before the return from the call in statement number (4).
   5. The stack at statement (7)

b) Describe how the address of each of the following variables can be pre-calculated by a compiler:
   1. *data* and *limit* in the main program

2. *limit* in function *Getlimit*
3. *i* in procedure *Initialize* as an l-value and an r-value
4. *s* as an l-value and r-value, *a*, and *m* in procedure *Add*

c) Generate code for the VC machine described on the last page for each of the following
   1. procedure *Add;*
   2. the *call to procedure Add* in statement (6)

d) Write Pascal code that compiles (parses and generates code) *SPL while statements* for the VC machine described on the last page.

II. **(33%)**

a) What is meant by intermediate representation, why and when would it be useful? Describe two different methods for expressing intermediate representations.

b) Consider the following Pascal code

```
sum:=0;
for i := 1 to 100 do begin
    read (x);
    sum:= sum+x;
end;
write x;
```

1. Give 3 address code for the above *for loop*. Use the following psuedo-instructions.
   JumpNeg x L (jump if the value of x is negative to label L)
   Jump L (jump unconditionally to label L)
2. Give triples, indirect triples and quadruples representations for the 3-address code obtained in part 1 above.

c) Give short circuit three address code for the following boolean expression
   (x<y) or (z< w) and (e<f)

III. **(29 %)**

a) A compiler checks that a source program follows the syntactic and semantic conventions of a programming language. Describe three semantic static checks the are typically performed by a compiler.

b) Despite the static checkings done by compilers, errors still occur. Why? Describe two kinds of errors that cannot be prevented by a compiler.

c) Consider the following translation scheme for determining types of expressions

```
E → literal      { E.type := char}
E → num          {E.type  := integer}
E → id           { E.type  := lookup(id.type) }
E → E 1+ E2      { E.type := if E1.type = integer and E2.type=integer
                       then     E.type := integer
                       else     type-error
```

1. Extend the above translation scheme to include the type *real* and to perform appropriate type coercion whenever required
2. Introduce an attribute *val* to hold the value of expressions and modify the given translation scheme so that it returns the value of an expression in *val* . How would you classify the attributes *val* and *type* in the resulting scheme. Explain
3. Modify the given translation scheme to print a descriptive message when a type error is detected and to continue the type checking as if the expected type has been seen.

IV. **(23%)**

a) Show that no left recursive grammar can be LL(1).

b) Prove or disprove: Every LR(1) grammar is LL(1).

c) Assuming a grammar G, an input string w and a parsing table with functions *action* and *goto*, give the general LR parsing algorithm for parsing w.

V. **(18%)**

a) When teaching introductory programming in Pascal, students are typically advised to pass arrays as var parameters. Why would that be useful? Is this advice equally good for programming arrays in C? Why?

b) Some programming languages do not reserve keywords. How does this affect the ease of writing a compiler for the language and why?

c) An innovative language designer decided to modify the definition of the Pascal programming language so that variables may be used before being declared? Comment on the practicality of this design from a compiler writer's perspective.

VI. **(16%)**

a) How many types does the SPL language have? What are these types?

b) Does your compiler perform any type coercion? Explain

c) How does SPL differ from Pascal regarding the convention of passing arrays as parameters?

d) How would you classify your compiler in terms of passes and parsing techniques?