CORE EXAMINATION
Department of Computer Science
New York University
January 2001

a) Write your name on the front of your exam envelope, in the blank space next to your exam number, and nowhere else. Print your SID underneath your name.

b) All booklets are labelled with a specific question number and your exam id number. Use the right booklet for each question!

c) Keep the slip of paper with your exam number. You'll need to know this number for the afternoon Algorithms exam, and for identifying your exam results when they are announced.

Programming Languages and Compilers

I) Consider a simple language with the following non-terminals:

```
stat-seq   : a sequence of statements
if-stat    : if-statement, both branches have a sequence of statements
while-stat : a loop construct, whose body has a sequence of statements
func-def   : a function definition, whose body is a sequence of statements.
ret-stat   : a return statement
any-stat   : any of the above, assignment, goto, etc.,
```

Most languages have the semantic requirement that a function body have at least one return statement somewhere.

a) For the non-terminals above, write the productions that enforce this rule, namely that a function definition contain **at least** one return statement. (hint: You will want to introduce additional non-terminals).

ANSWER. a) We start with the following grammar:

```
stat-seq   -> {statement}*
if-stat    -> IF expr THEN stat-seq ELSE stat-seq END
while-stat -> WHILE expr DO stat-seq END
ret-stat   -> RETURN expr
any-stat   -> any statement
```

We now introduce a new non-terminal, which is a sequence of statements that must have **at least** one return statement:

```
ret-stat-seq -> stat-with-return stat-seq |
                any-stat ret-stat-seq
```

Now a statement with a return is one of the following

```
  stat-with-return -> ret-stat | ret-if-stat | ret-while-stat

A ret-if-stat is an if statement that has at least one return

  ret-if-stat -> IF expr THEN ret-stat-seq ELSE stat-seq END  |
                 IF expr THEN stat-seq     ELSE ret-stat-seq END

And similarly, a ret-while-stat is a while with at least one return

  ret-while-stat -> WHILE expr DO ret-stat-seq END
```

Now a function is defined as having a ret-stat-seq instead of a stat-seq, and we know there is at least one return somewhere in its body.

b) Explain why defining this restriction as a syntactic (context-free) rule is not a good idea. How would such a rule typically be enforced in a compiler?
ANSWER. There are two problems with trying to do this syntactically. First the grammar above is ambiguous. Intuitively, if you see a sequence of statements, it can be a stat-seq, as well as a ret-stat-seq. It may be possible to make an unambiguous grammar, but probably at the expense of many more productions. From a parsing point of view that's not good enough, one also needs a grammar that can be parsed without backup, and as written the grammar appears to be non-deterministic.

Second, even if the grammar could be fixed to be deterministic, there is no gain in detecting this problem at parse time, and a parser that used this grammar would typically diagnose a function without a return with an totally uninformative message ("expect legal function body") indicating that all of the productions for a function had failed.

It is always better to leave checks like this to the semantic analysis phase if the check is easily done there. That's most certainly the case here, where it is easy enough to have a boolean flag that is set when we process a return statement (the semantic phase has to traverse the tree). Not only is this much easier to program, but it is much easier to give a clear error message. Much better to add one small check to the semantic phase, than complicate the whole grammar for this one rule.

II) Consider the following C++ class:

```
class phone {
   char* name;
   char* number;
   public:
      phone () { name = 0; number = 0;}
      phone (char* who, char* what) {
         name =   new char [strlen (who)]; strcpy (who, name);
         number = new char [strlen(what)]; strcop (what, number);
}
```

a) Write a proper destructor for this class.

ANSWER. An object of the class allocates two dynamic structures, those are the ones that must be reclaimed:

```
~phone () {
    if (name != 0)  delete  name;
    if number != 0) delete number;
}
```

Note that the guard is necessary, because there is no guarantee that the object was built with the second constructor.

b) Suppose we now declare:

```
{
 phone my_book[5] = ....

 ...  // some code to handle my phone book.
 }
```

Explain how the storage allocated for the entries in my_book gets reclaimed.

ANSWER. The destructor is applied to each component of the array. The user does not have to write a destructor, the compiler generates the loop that invokes the destructor over the known range of the array. The array itself is local and allocated on the stack, and does not require any reclamation.

c) Java does not have the concept of destructor. Explain why.

ANSWER. In Java all storage management is out of the hands of the programmer, and the garbage collector takes care of storage reclamation. However, destructors play an additional role, namely returning resources to the system when execution ends abnormally, i.e. when an exception is raised. In that case, the destructors are invoked before the exception is propagated, so that for example files that were opened locally can be closed. In Java this is programmed by adding a **finally** clause to a **try** block, so that this is in fact an explicit user-supplied destructor for a scope.

III. Consider the following ML function:

```
fun meld proc lis1 lis2 = if null lis1 then nil
        else proc (hd lis1, hd lis2) :: meld proc (tl lis1) (tl lis2);
```

In words: meld takes a function and two lists, applies that function to corresponding elements of both lists, and returns the list of the results.

a) Write the most general type expression that describes the behavior of meld.

ANSWER. There is no requirement that lis1, lis2, and the result list have the same component types. So there are three type variables here, and the most general type of meld is:

```
('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

In words, lis1 is a list of a's, lis2 is a list of b's, proc takes an a and a b to yield a c, and the result is a list of c's.

b) In the imperative language of your choice (Java, C++, Ada), write a software component (function, template, or generic unit) that has the same functionality. Explain how your component can work with lists of different types. Explain why your solution is (or is not) type-safe.

ANSWER. We have three type variables, so a fully parameterized solution requires three type parameters. The function proc is also a parameter of the resulting package or method. In addition, we have to decide how to represent the lists in our chosen language. Assuming that lists are an existing parameterized class, we can write the following in C++:

```
template <class T1, class T2, class T3> class meld {
   typedef &T3 (*PF) (&T1, &T2);   // the type of the functional argument

   public &list <T3> meld (FP proc, &list<T1> lis1, &list <T2> lis2) {
   //  the algorithm is identical: apply proc to corresponding elements
   //  of lis1 and lis2, recurse, and concatenate.
  }
```

In Ada, we would write:

```
generic
  type T1 is private;
  type T2 is private;
  type T3 is private;
  with function proc (X : T1; Y : T2) return T3;
package Meld_Pack is
   type L1 is array (integer range <>) of T1;  -- for a change.
   type L2 is array (integer range <>) of T2;
   type L3 is array (integer range <>) of T3;
   function Do_Meld (Lis1 : L1; Lis2 : L2) return L3 is
   begin
      --  same algorithm.
   end Do_Meld;
end Meld_Pack;
```

Note that in this case, the lists (represented as array types) are exported by the instance, rather than being parameters of the generic. One might argue that a generic where the array types are also generic parameters is more flexible, because the list type can then exist before the instance is constructed.

Finally, in Java we can use predefined containers for objects (vectors for example), in which case there are no parameterized types at all. To pass the procedure as a parameter, we define an interface with a single method, and pass an instance of that interface to the constructor for Meld. In contrast to the two other solutions, the Java approach is not type-safe: if the lists are lists of objects there is no way to verify statically that they are homogeneous, or that their components are compatible with the expected types of of the arguments of proc. The body of proc will have to have casts from Object to the expected types, and these casts will have to be checked at run-time. The Ada and C++ solutions guarantee that the lists are homogeneous and that they match the requirements of proc.

c) What happens in your component if the two lists have different lengths? what is the best way of specifying the behavior in this case?

ANSWER. The code should verify that both lists are of the same length, and throw an exception if not. The ML solution is incomplete in that respect: if lis1 in null it does not check lis2. Otherwise an exception will be thrown automatically during recursion, when **tl** is applied to an empty list, if lis2 is shorter than lis1.

Depending on the implementation in an imperative language, applying the tail selector to an empty list might or might not raise an exception. Good programming practice is to make sure that an exception is raised, and **not handled in the procedure itself** but propagated to the caller. Printing a message is useless, it is the caller code that must be notified, and must be prepared to handle the exception. In Java, the methods should be declared as throwing a particular exception, so that the caller is forced to write a handler for it.