CS 538

Homework #1

Due: Monday, February 25, 2002

(Not accepted after Monday, March 4, 2002)

1. Most procedural programming languages, including C, C++ and Java, execute statements **sequentially**. That is, a sequence of statements S_1; S_2; ...; S_n (separated by ";") specifies that statement S_1 be executed, then S_2, and finally S_n.

   Some programming languages (like Algol 68) provide an alternate to sequential execution—**collateral** execution. A sequence of statements S_1, S_2, ..., S_n (separated by ",") means execute statements S_1 through S_n **in any order**. Thus S_2 might be executed first, then S_4, then S_1, etc. Eventually all n statements are executed, but the exact order is unspecified by the programmer.

   (a) What advantages (if any) are there in using collateral execution rather than sequential execution in a program?

   (b) Is collateral execution always equivalent to sequential execution? That is, for any set of statements S_1 to S_n (in C or Java—your choice), does S_1; S_2; ...; S_n and S_1, S_2, ..., S_n always compute the same result?
   If so, explain carefully why. If not, give a simple example that demonstrates the inequivalence.

   (c) If your answer to (b) was that sequential and collateral execution are not always equivalent, explain the circumstances under which collateral and sequential execution are equivalent. That is, what properties must S_1 to S_n have to allow either execution mechanism to be used (without changing what is computed)?

2. In the early days of programming language design, procedure calls were explained using a **macro-expansion** model. That is, the effect of a procedure call $P(e_1,e_2,...,e_n)$ was defined to be equivalent to expanding the body of P at the point of call, with all occurrences of P's first formal parameter replaced by e_1, P's second formal parameter replaced with e_2, etc. Any calls that appear within P were also modeled using macro expansion.

   Note that it isn't necessary to implement calls this way—we can simply use macro-expansion as a way to explain the effect of a procedure call.

   (a) A procedure body often contains references to identifiers that are not defined locally. Such identifiers are said to be **free** (because they are not bound to identifiers defined within the procedure body. For example, in the following simple C procedure, identifier c is free:

```
int p(int a) {
   int b;
return a+b+c; }
```

Recall that a free variable in a procedure can be bound either statically or dynamically. Which binding method should be used if the macro-expansion model of calls is followed? Explain why.

(b) A number of parameter passing mechanisms, including call by value, call by reference and call by name have been used in programming languages. Which of these (if any) correspond to the macro-expansion model of calls? For each parameter passing mode you should explain carefully why it corresponds to the macro-expansion model, or give a simple example illustrating why it fails to match that model.

3. Programmers have long known that the effect of various data structures can be "simulated" using function calls. Thus rather than use a variable `current_date`, we can call a function `get_current_date()`. Similarly, instead of using an array, we can call a function that computes the array's value at each index. Instead of fetching the value of a field in a class or struct, we can lookup the value associated with a field name in a hash table.

   A limiting factor in use of functions to simulate variables is that we can't assign or update the value of a function. Thus while we can use `a[i]=x;` to update `a`'s value at index `i`, we can't use `f(i)=j;` to change `f`'s return value for parameter value `i`. But what if we could?

   (a) Give a reasonable definition of what it should mean to execute `f(i)=j`; in a language like C or Java (your choice). Would it be possible to implement your definition with reasonable efficiency? Would the ability to assign to functions be a useful addition to C or Java?

   (b) What if we allowed programmers to assign functions of zero arguments rather than values to function values? That is, we would now allow `f(i)=fun() { ... };` where after the assignment `fun() { ... }` would be executed whenever `f(i)` is called. Would this be a useful addition to C or Java? Could it be implemented effectively?

4. Many programming languages allow creation of a new name for an existing data type using some form of type declaration. In C and C++, `typedef` can be used:

   ```
   typedef  int  integer;
   ```

   defines `integer` to be a synonym for `int`.

   Assume that we wish to take an existing data type, like `int` or `float`, and create a brand new type that inherits all of the literals, operators and library subroutines of the old type, but which is **type-inequivalent** to it. For example, we might use

   ```
   newtype  meters is created from float;
   ```

   to create a type `meters` that is different from `float`, but which has literals like `2.1` or `10e-5`, operators like `+` or `*`, and library routines like `sqrt` or `toString`.

   What type rules are needed to determine if a program that uses both an original type (like `float`) and a newly created type (like `meters`) is type-correct?.

5.(a) The C programming language has the operator & that returns the address of a variable. If the address of a variable is assigned to a pointer variable, is this address valid throughout the rest of program execution? If so, explain why. If not, explain what rules on assignment of variable addresses are necessary to guarantee that only valid addresses are used during execution.

(b) C++ uses the `new` operation to allocate heap objects. Heap objects must be explicitly freed using the `delete` operation.

Assume a call to `new` appears in procedure `P`. Under what circumstances can a call to `delete` be *automatically* added at the end of `P` to guarantee that the heap object is properly disposed of?

6. Structural equivalence in languages that contain structs (like C and C++) is defined as follows.

> Struct `S1` is structurally equivalent to struct `S2` if `S1` and `S2` contain the same number of fields and corresponding fields (in order of declaration) in `S1` and `S2` are structurally equivalent. (The names of corresponding fields within the two structs need not be the same)

Two pointers are structurally equivalent if the types they point to are structurally equivalent. That is, `S1*` is structurally equivalent to `S2*` if and only if `S1` is structurally equivalent to `S2`. Two arrays are structurally equivalent if they have the same size and their component types are structurally equivalent. Each scalar type is structurally equivalent only to itself.

(a) Assume a C-like language in which structs may contain fields declared to be scalars (`int`, `float`, etc.), arrays, pointers and structs. Give an algorithm that decides if two structs, `S1` and `S2`, are structurally equivalent.

(b) Most languages, including C and C++, state that the order in which fields are declared is unimportant. That is, rearranging field declarations in a struct has no effect other than possibly changing the size of the struct.

Given this observation, it might make sense to change the rule for structural equivalence of structs so that two structs are structurally equivalent if they contain the same number of fields and it is possible to reorder the fields of one struct so that corresponding fields are structurally equivalent after reordering. That is, the order of fields in a struct no longer matters (nor does the name of fields). Thus the following two structs are now considered structurally equivalent:

```
struct S1{
   int f1;
   float f2;
}

struct S2{
   float g1;
   int g2;
}
```

Update your algorithm of part (a) to implement this revised definition of structural equivalence. Illustrate your algorithm on the following set of structs; is S1 structurally equivalent to S2?

```
struct S1{
   S1* f1;
   S2* f2;
   S3* f3;
}

struct S2{
   S4* g1;
   S1* g2;
   S2* g3;
}

struct S3{
   S4* h1;
}

struct S4{
   S3* j1;
}
```